qte-doc Documentation

Release 0.5.0

David Townshend

September 06, 2012

Contents

qte is a wrapper around the PySide GUI toolkit making it simpler and more pythonic to use. No changes are made to the original PySide classes, but new classes and functions are introduced which effectively replace much of the most commonly used PySide GUI functionality.

CHAPTER 1

Key features

- All objects (i.e. those contained in `PySide.QtGui`, `PySide.QtCore`, and the new objects provided by this package) can be imported from a single namespace, e.g:

  ```
  >>> from qte import QRect, QWidget, DataModel
  ```

  Or, more commonly:

  ```
  >>> import qte
  >>> rect = qte.QRect()
  ```

  The PySide objects, i.e. those prefixed with "Q", are exactly the same as those imported directly from PySide, all changes are in the new classes.

- A number of convenience function had been added. See *Core Tools* for details.
- The PySide model/view framework has been extended and simplified. See *Model/View framework* for more information.
- A new singleton `Application` class is available, extending the functionality of `QApplication`. See `Application` and `runApp` for more information.
- A `Document` class is provided to manage typical document application functions, such as opening, saving, etc.

CHAPTER 2

Installation

qte can be installed from the cheeseshop, or downloaded directly from http://bitbucket.org/aquavitae/qte. It requires PySide >= 1.1.

Table of contents

3.1 Tutorial

3.1.1 Introduction

Qt is a highly complex and versatile toolkit, but this flexibility sometimes makes it difficult to formulate workflows. This short tutorial illustrates a suitable workflow for developing desktop applications with PySide and qte.

3.1.2 Hello World

All tutorials start with a "Hello World" example:

```
import qte
label = qte.QLabel('Hello World!')
qte.runApp(label, 'MyApp')
```

And here is an explanation of each line.

1. `qte` is imported. All of PySide's `QtCore` and `QtGui` classes, can be accessed directly from this namespace, as well as the extra objects provided by `qte`.
2. A new `QLabel` is created from the `qte` namespace.
3. The application is launched with the label as the main window and the application name as 'MyApp'.

The most important part of this example is the `qte.runApp` function. This is essentially equivalent to:

```
qte.Application().setApplicationName('MyApp')
qte.Application().setMainWindow(label)
win.setWindowTitle('MyApp')
win.show()
qte.Application().exec_()
sys.exit()
```

Note that `qte.Application` is used instead of `QApplication`. The `qte.Application` class provides a few extra features which make it more suitable to desktop applications, such as settings management. It is also a singleton which on its first initialisation calls `QApplication([sys.argv])`.

3.1.3 UIs and Resources

Qt allows for two ways of designing user interfaces; they can either be hard coded or created using Qt Designer. In practice, it is common for a combination of these methods to be used. When coding Qt in C++, the normal

workflow consists of designing the UI and creating resources in Qt Designer, then compiling them into binary files which can be inserted into the executable. In python, code is seldom compiled to an executable at all, so compiling *ui* and *qrc* files becomes a rather annoying and tedious exercise. `qte` provides an alternative may of dealing with these.

Resources can be compiled at runtime using `qte.loadResource`. The binary data created by this is identical to the output of *rcc*, but the function is implemented purely in python, with no dependency on *rcc* or *pyside-rcc* at all. If used with the *register* argument, it also registers the data with the resource system so that resources can be used immediately.

PySide's `QUiLoader` class can be used to create widgets at runtime from *ui* files. `qte` extends this with `qte.loadUi` and `qte.uiWrapper`. `qte.loadUi` does the same job as `QUiLoader.load`, but first registers custom widgets and resources. `qte.uiWrapper` wraps the widget in another class and is especially useful for `QMainWindows` which cannot be promoted in Qt Designer.

The following example shows how to load and inherit from a `QMainWindow` interface created in Qt Designer. The window has a single button called `showDialog` which, when clicked, loads and displays a dialog from another *ui* file with itself as the parent. The button icon is read from a resource file:

```
import qte

class MainWindow(qte.UiWrapper('ui/mainwindow.ui'):

    def __init__(self):
        qte.loadResource('icons.qrc', register=True)
        self.showDialog.setIcon(':dialog.png')
        self.showDialog.clicked.connect(self.loadDialog)

    def loadDialog(self):
        dialog = qte.QUiLoader().load('ui/dialog.ui', self)
        dialog.show()
```

3.1.4 Model/View Programming

One of the main problems with Qt's model/view framework in a python environment is the assumption that data is stored, or only visible through a `QAbstractItemModel`. In C++ this is an ideal structure for storing structured data, but in python lists and dicts provide more flexibility. `qte.DataModel` gives this flexibility by wrapping a `QAbstractItemModel` interface around generic python structures. Currently it only supports tabular data (i.e. which can be displayed in a `QTableView`), but in the future tree-like structures will be supported too.

An `qte.DataView` class is inherited from `QTableView` and has a few extra features and customised defaults.

There are also several new delegates, all based on `qte.TypedDelegate` which supports more data types than the default `QItemDelegate`.

3.2 UIs and Resources

PySide's resource system relies on compiling all resource files before using them, which goes against the normal python workflow. To improve upon this, `qte` provides functions to dynamically load resources and user interfaces.

3.2.1 Working with Resources

Resources should be handled in a similar with `qte` as with Qt; all resources should be stored in a location referenced by a *qrc* file. In Qt, this would usually be compiled into a binary resource file when the code is compiled. In `qte`, it is preferable not to compile anything, so the files are read directly. The `loadResource` function parses a *qrc* file and loads all referenced resources into the Qt resource system, so that they can be loaded using normal Qt syntax (e.g. `QIcon(':myicon.png')`).

3.2.2 Working with UIs

ui files created with Qt Designer can be loaded in several different ways, depending on their usage. The simplest way is to create a new instance of the widget defined in the file by calling `loadUi`. However, if the widget requires further customisation then it is preferable to create a new widget class rather than an instance. This can be done with the `uiWrapper` function.

3.2.3 API

`qte.`**`loadResource`**(*qrc*[, *register=True*[, *split=False*]])
Compile the resource file *qrc*.

Parameters

- **qrc** – The name of a *qrc* file.
- **register** – If `True` (default), register the resource data with Qt's resource system.
- **split** – If `True`, return tuple of `(tree, names, data)`

Returns A `bytes` object containing the raw, uncompressed resource data.

The raw resource data is returned, and can be written to a file to fully mimic Qt's *rcc* tool. However, in most cases this is not required and can be ignored.

If *split* is `True`, a tuple of `(tree, names, data)` is returned. Each of theses is a `bytes` object and is the same as the data contained in *qt_resource_struct*, *qt_resource_name* and *qt_resource_data* respectively, in a file created with *pyside-rcc*. This is mainly useful for debugging.

`qte.`**`loadUi`**(*ui*[, *parent*[, *widgets*]])
Create a new instance of the widget described in *ui*.

Parameters

- **ui** – The name of a *ui* file.
- **parent** – The parent widget, or None.
- **widgets** – A list of custom widgets which are reference in *ui*

Returns A widget instance.

This uses `QtUiTools.QUiLoader` to create the widget, but first registers all custom widgets in `qte` and those explicitly set in *widgets*, so these widgets can be used in Qt Designer by promoting them from their base classes. Note that the header file field in Qt Designer can be set to anything.

`loadUi` calls `loadResource` to register all resource files specified in the *ui* file.

`qte.`**`registerWidget`**(*name*[, *widget*])
Register a widget referenced in a *ui* file.

If *widget* is omitted, then a class decorator is returned, allowing it to be used like this:

```
@registerWidget('MyWidget')
class MyWidget(QWidget):
    ...
```

`qte.`**`uiWrapper`**(*ui*[, *parent*])
Return a class inherited from a widget defined in a *ui* file.

This is typically used when customising a widget created using Qt Designer:

```
class MyWidget(uiWrapper('mywidget.ui')):
    def __init__(self, value):
        super().__init__()
        self.value = value
```

Resource files referenced by the *ui* file are automatically loaded using `loadResource`.

There are several opportunities for naming clashes, and understanding these requires a knowledge of how the working globals are used.

- The globals dict is first populated with all available widgets, i.e those provided by PySide, `qte` and registered using `registerWidget`.
- When loading the *ui* file, a special class is created named `UI_<widget_name>`, which will override any similarly named widgets.
- When the class is first initialised, all widgets added to in in Qt Designer are added to its dictionary.

To avoid the potential pitfalls, ensure that registered widget names do not start with `Ui_` and that objects added in Qt Designer do not have the same name as a parent widget property. For example, do not add a child widget named "objectName".

3.2.4 Resource file reference

The specification for rcc files is not documented, but from a study of the rcc source code, it appears that this is the format: The file is binary and consists of four sections, in order.

Header

This is general information about the file.

Position	Size	Value	Description
0	4	"qres"	Magic number
4	4	0x01	RCC version. This is always 0x01
8	4	t_off	Position of start of Tree section (usually 20)
12	4	d_off	Position of start of Data section (= n_off + len(names))
16	4	n_off	Position of start of Name section (= t_off + len(tree))

Tree

The Tree section contains information about the relative positions of files and directories in the resource tree. It is a sequence of records, sorted by the hash of the node name in each branch. For example:

root
- a
- b
  - -a
  - -c
- -c

The structure of each node depends on whether if references a file or a directory. For directories (including root):

Position	Size	Value	Description
0	4	n_off	Offset of name record in the Names section
4	2	flags	A bitwise OR combination of flags. 0x1 indicates a compressed file. 0x2 indicates a directory.
6	4	c_child	The number of children nodes in the tree. The children records will follow this node.
10	4	c_off	The index position of the node in the list of nodes. *root* is always 1 and they are numbered sequentially, including file nodes (although the number is not recorded for file nodes).

For files:

Position	Size	Value	Description
0	4	n_off	Offset of name record in the Names section
4	2	flags	A bitwise OR combination of flags. 0x1 indicates a compressed file. 0x2 indicates a directory. Normal files are either 0x0 or 0x1.
6	2	l_cntry	The locale's country code as specified in `QLocale`. The locale defaults to C.
8	2	l_lang	The locale's language code as specified in `QLocale`. The locale defaults to C.
10	4	c_off	The node's offset in the Data section.

Names

This section contains a list of node names, in the same order as they appear in the Tree section.

Position	Size	Value	Description
0	2	c_name	The length of the name.
4	8	h_name	The hash of the name (calculated using `qHash`).
8	2 * c_name	name	The name, encoded in unicode as 2-bytes per character.

Data

This is the actual data in the resources and only applies to files, not directories.

Position	Size	Value	Description
0	4	l_data	The length of the data.
4	l_data	data	The data contained in the resource, compressed if the appropriate flag is set in the Tree section.

3.3 Edit Widgets

PySide has a range of simple, one-line editing widgets with slightly different APIs. In order to make it possible to program type-independantly, several of these have been subclassed to conform with `EditWidgetABC`.

3.3.1 EditWidgetABC

class `qte.`**`EditWidgetABC`**

Define the API of a standard editing widget.

EditWidgets supports a specific set of types, defined by its `types` class method. They may also handle and return `None`, which indicates an invalid value. It is up to the subclass implementation to deal with type checking and conversions. While all of these classes support setting the value to None, they do not necessarily have to return None. This behaviour depends on the specific class.

valueChanged
This signal is emitted whenever the value changes.

It is similar to the `textChanged` signal in a `QLineEdit`.

emitValueChanged()
Called by subclasses to raise the `valueChanged` signal.

getValue()
Return the value of the widget. This is similar to `QLineEdit.text`.

setValue(*value*)
Set the value of the widget. This is similar to `QLineEdit.setText`.

classmethod types()
Return a list of types supported by the widget.

3.3.2 CheckBox

class qte.**CheckBox**([*value=False*, *parent=None*])
A standard check box implementing `EditWidgetABC`

Inherits: `QLineEdit`
Datatype: `bool`

`None` is an acceptable value for `setValue` and `getValue` and indicates that the check box is partially checked.

3.3.3 ComboBox

class qte.**ComboBox**([*value=''*, *parent=None*])
An editable combo box implementing `EditWidgetABC` which reports on the current text.

Inherits: `QComboBox`
Datatype: `str`

`None` is an acceptable value for `setValue` and is converted to an empty string. `None` is returned by `getValue` if no index is selected.

3.3.4 DateEdit

class qte.**DateEdit**([*value=None*, *parent=None*])
A standard date edit widget implementing `EditWidgetABC`.

Inherits: `QDateEdit`
Datatype: `datetime.date`

`None` is an acceptable value for `setValue` and is converted to the default date used by `QDateEdit`. `None` is never returned by `getValue`.

3.3.5 DateTimeEdit

class qte.**DateTimeEdit**([*value=None*, *parent=None*])
A standard date and time edit widget implementing EditWidgetABC.

Inherits: QDateTimeEdit
Datatype: datetime.datetime

None is an acceptable value for setValue and is converted to the default value used by QDateTimeEdit. None is never returned by getValue.

3.3.6 FloatEdit

class qte.**FloatEdit**([*value=None*, *parent=None*, *decimals=None*])
A standard line edit widget for floats implementing EditWidgetABC.

Inherits: QLineEdit
Datatype: float

If given, *decimals* sets the number of decimals to round the value to.

None is an acceptable value for setValue and is converted to an empty string. None is returned by getValue if the value cannot be converted to a float.

3.3.7 IntEdit

class qte.**IntEdit**([*value=None*, *parent=None*])
A standard line edit widget for integers implementing EditWidgetABC

Inherits: QLineEdit
Datatype: int

None is an acceptable value for setValue and is converted to an empty string. None is returned by getValue if the value cannot be converted to an integer.

3.3.8 OptionsBox

class qte.**OptionsBox**([*value=None*, *parent=None*])
A combo box which allows a single selection from a list of options.

Inherits QComboBox
Datatype: SelectList

Inherits QComboBox
Datatype: SelectList

None is an acceptable value for setValue and is converted to an empty SelectList. None is never returned by getValue. The SelectList passed to setValue is copied and the selected option tracked by index. This means that non-string types in setValue will be returned by getValue intact.

3.3.9 TextEdit

class qte.**TextEdit**([*value=''*, *parent=None*])
A standard line edit for strings implementing EditWidgetABC.

Inherits: QLineEdit
Datatype: str

None is an acceptable value for setValue and is converted to an empty string. None is never returned by getValue.

3.4 Model/View framework

PySide, through Qt, has an extensive and intricate model/view framework. This package provides a simpler interface to this through a single model class, DataModel and a single view class DataView which are built on top of QAbstractTableModel and QTableView respectively but which support tabular data structures. A HideProxyMixin class can be used as a mixin for proxy models to automatically map unrecognised attribute lookups to source models. Two new proxy models, SortFilterProxyModel and AppendProxyModel use this mixin.

3.4.1 DataModel

The DataModel works on the premise that the data itself is stored and managed in some sort of python structure, for example a list. The data is assume to be essentially tabular with defined columns. The most important method is source, which returns an iterator over source records and is internally cached by the model to improve performance. source can be re-implemented through inheritance, or by simply assigning the name to a function, e.g.:

```
mymodel.source = lambda: iter(mydata)
```

The default implemented returns the same list every time, so it may be used to assign static data, e.g.:

```
mymodel.source().append(['new row'])
```

Each row in the table is represented by an item returned by source. The only requirement of the source items are that they should also be iterable.

Columns are defined by setting a list of column titles to DataModel.titles.

class qte.**DataModel**([*titles*])
Create a new DataModel with a list of column titles.

This inherits from QAbstractTableModel and implements columnCount, rowCount, flags, headerData, data and setData.

cache_timeout
Minimum number of seconds between cache refreshes (default = 1).

titles
A list of column titles.

cache_refresh()
Force a cache refresh now.

columnCount(*[parent]*)
Return the number of columns as determined from `titles`.

parent is superfluous and is ignored. It is only provided for Qt compatibility.

data(*index*, *role*)
Return the data in `source` for `DisplayRole` and `EditRole`.

flags(*index*)
Return `Qt.ItemFlags` for a cell.

By default, all cells are selectable, editable and enabled.

See Also:

`setFlags`

headerData(*section*, *orientation*, *role*)
Return header information.

For horizontal headers and `Qt.DisplayRole`, return the relevant item in `titles`.

record(*row*)
Return the record currently appearing on a *row*.

Parameters **row** – Integer row number.

Returns A record

This takes into account filtering and sorting on the model.

rowCount(*[parent]*)
Return the number of visible rows under *parent*.

parent is superfluous and is ignored. It is only provided for Qt compatibility.

setData(*index*, *value*, *role*)
This has been re-implemented to call `setValue` for `EditRole`.

setFlags(*column*)
A convenience method to set *flags* for *column*.

Flags may be set either as an bitwise-or combination of `Qt.ItemFlags` or as a space-separated list of any of the following strings.

String	Qt.ItemFlag	Description
selectable	ItemIsSelectable	It can be selected.
editable	ItemIsEditable	It can be edited.
drag	ItemIsDragEnabled	It can be dragged.
drop	ItemIsDropEnabled	It can be used as a drop target.
checkable	ItemIsUserCheckable	It can be checked or unchecked by the user.
enabled	ItemIsEnabled	The user can interact with the item.
tristate	ItemIsTristate	The item is checkable with three separate states.

setValue(*record*, *column*, *value*)
Set the value in *record* for a specific *column* to *value*. By default, this assigns *value* using `record[column] = value`. This should never be called directly, or the `Qt.dataChanged` signal will not be emitted.

Parameters

- **record** – A record in *source*.
- **column** – The column number.
- **value** – The value to store in the record.

Returns `True` if the value was stored successfully.

source()
Return an iterator over source data that this model represents. The default implemention returns the same list on every call.

value(*record*, *column*)
Return the value stored in *record* for a specific *column*. By default, this uses index lookup on the column number, i.e `record[column]`.

Parameters

- **record** – A record in *source*.
- **column** – The column number.

Returns The value stored in the record.

3.4.2 CheckFilterModel

class qte.**CheckFilterModel**([*source*[, *titles*]])
A list model presenting filter options.

This subclasses from `DataModel`, and has a single column containing `"Select All"` and the values in `source`. `"Select All"` is tristate, indicating (and setting) the check state of all other values.

If *titles* is omitted, a single column is assumed.

This would commonly be used as the model of a combo or list widget to allow the user to filter out values from a data source. By default, all items are selected and any new ones added are automatically selected.

When the filter changes, a filterChanged signal is emitted with a set of unselected values.

filterChanged(*set*)
This signal is emitted when the checkstate of any item changes.

The value is a set containing the unselected values in the list.

selectedState()
Return the check state of the `"Select All"` option

3.4.3 Proxy Models

A new mechanism is provided to hide additional proxy layers between the view and the source model by use of a `HideProxyMixin` mixin class. Two new proxy classes are also available: `AppendProxyModel` and `SortFilterProxyModel`.

class qte.**HideProxyMixin**
This mixin class provides access to source model attributes and methods.

The following example illustrates usage on a custom model:

```
class SortModel(QSortFilterProxyModel, HideProxyMixin):
    pass

mdata = QStandardItemModel()
msort = SortModel()
msort.setSourceModel(mdata)
```

This makes the following two calls identical:

```
item = msort.itemFromIndex(proxyindex)
item = msort.sourceModel().itemFromIndex(msort.mapToSource(proxyindex))
```

Arguments and return values may be converted if required, e.g. mapping of model indexes. The conversions used are based on annotations in the source model, which should be one of `'row'`, `'column'` or

`'index'`. The conversions are done by methods in the proxy model *mapRowFromSource*, *mapColumnFromSource*, *mapFromSource* and the corresponding *ToSource* methods. *mapToSource* and *mapFromSource* are defined by PySide. The others are proxided by the mixin, but may be re-implemented.

`SortFilterProxyModel` and `AppendProxyModel` use this mixin.

mapRowToSource(*row*)
Map a row to the source model.

This simply calls `mapToSource` with an index in column 0, and should be re-implemented when a more direct method can be used.

mapColumnToSource(*column*)
Map a column to the source model.

This simply calls `mapToSource` with an index in row 0, and should be re-implemented when a more direct method can be used.

mapRowFromSource(*row*)
Map a row from the source model.

This simply calls `mapFromSource` with an index in column 0, and should be re-implemented when a more direct method can be used.

mapColumnFromSource(*column*)
Map a column from the source model.

This simply calls `mapFromSource` with an index in row 0, and should be re-implemented when a more direct method can be used.

class `qte.`**AppendProxyModel**([*parent*])
This proxy model provides an interface to append rows.

For the most part, data is mapped straight the source model. However, this model always provides an additional empty row at the end which can be used to enter new data. Every time the data is changed in this row, the `appendDataChanged` signal is emitted. The source model is expected to emit `rowsInserted` if the append data was accepted, so if the source model emits this signal to indicate appending a single row, the pending data is cleared.

This class inherits from `HideProxyMixin`, and provides all the expected mapping functions.

defaults
A dict containing default value to use when editing pending data.

Each value in this dict should be keyed by the column number. If no default is set, `None` is used.

appendDataChanged(*dict*)
This signal is emitted whenever the pending data changed.

The argument is a dict of values set keyed by column number. This signal is normally connected to a slot in the source model, which is expected to emit `rowsInserted` if the data is accepted.

clear([*column*])
Clear pending data by column.

If column is omitted, then clear everything.

data(*index*[, *role=Qt.DisplayRole*])
Return data at *index* for *role*.

If *index* refers any row except the pending row, the source model data is returned. For the pending row, the return value is as follows:

Role	
BackgroundRole	A brush using `QPalette.Midlight`
EditRole	Pending, default or `None`, depending what has been set.
DisplayRole	Pending or an empty string, depending what has been set.

flags (*index*)
Return the flags for an index.

Return the source model's flags for all but the last row. The last row returns `ItemIsEnabled` and `ItemIsEditable`.

headerData (*section*, *orientation*, *role*)
Return header data from the source model.

In addition, the vertical header at the append row has a "clear" icon. The `sectionClicked` signal for this header should be connected to `clear` for it to work. This is done automatically if used with `DataView`.

setView (*view*)
Called when the model is set to a `DataView`.

unsetView (*view*)
Called when the model is removed from a `DataView`.

class `qte.`**SortFilterProxyModel**
Applies per column filtering to QSortFilterProxyModel.

This behaves almost exactly the same as `QSortFilterProxyModel`, but allows separate filters for each column. Also, the filters are callables which take a value and return *True* (display) or *False* (hide).

This class inherits from `HideProxyMixin`, and provides all the expected mapping functions.

filterFunction(column):
Return the filter function and role as set by `setFilterFunction`.

The return value is a tuple of `(function, role)`. If no filter has been set, both values are `None`.

setFilterFunction (*column*, *func*[, *role=qte.Qt.DisplayRole*])
Assign a filter function to a column.

role sets which role is used for obtaining the values passed to *func*. For example, to hide all rows with a red background in the second column:

```
def filter_function(value):
    if isinstance(value, QBrush):
        return value.color() != qte.QColor(QColor.red)
    else:
        return True
proxy.setFilterFunction(1, filter_function, qte.Qt.BackgroundRole)
```

3.4.4 Delegates

The basic delegate is `TypedDelegate`, which intelligently manages several data types, including `datetime` and `SelectList` values. Delegates for specific types are subclassed from this, and may be used explicitly instead for better control.

class `qte.`**TypedDelegate** (*parent*)
A delegate which handles various python types.

Subclasses should provide at least an implementation of `getWidget`, which returns a new instance of a widget to use for editing. The delegate removes the widget frame for better style consistency.

The default implementation guesses the widget based on the data type returned from the model by `EditRole`, using `guessWidget`. If the expected data type is known, however, it is better to use one of the strictly typed delegates, `FloatDelegate`, `DateTimeDelegate` or `ListDelegate`.

getWidget (*parent*, *option*, *index*)
Return the widget to use for editing.

class `qte.`**DateTimeDelegate** (*parent*[, *fmt*, *usetime=False*])
A delegate for `datetime` values.

Parameters

- **fmt** – A format string for display as used by `datetime.datetime.strftime`. If it is omitted then the system default is used.
- **usetime** – A boolean specifying whether the time is managed by the delegate. If `True`, then the delegate uses `datetime.datetime`. If false, it uses `datetime.date`.

getWidget (*parent*, *option*, *index*)
Return a `DateEdit`, or `DateTimeEdit` if *usetime* is `True`.

class `qte.`**FloatDelegate** (*parent*[, *prefix=''*, *decimals=None*, *suffix=''*])
A delegate for float values.

Parameters

- **prefix** – A text prefix to add to numbers displayed, and strip from numbers entered.
- **decimals** – The number of decimal places to use when displaying the number. If omitted, then all decimals are shown.
- **suffix** – A text suffic to append to numbers displayed, and strip from numbers entered.

For example:

```
>>> widget = QWidget()
>>> dg = FloatDelegate(widget, '$ ', 1, ' million')
>>> dg.displayText(12.345678, None)
'$ 12.3 million'
```

getWidget (*parent*, *option*, *index*)
Return a FloatEdit with *decimals* set as in the constructor.

class `qte.`**ListDelegate** (*parent*)
A delegate for option lists.

When using this delegate, the model's `data` method is expected to return a `SelectList` object for `EditRole` and a string for `DisplayRole`.

getWidget (*parent*, *option*, *index*)
Return an `OptionsBox`.

3.4.5 Views

class `qte.`**DataView** ([*parent*])
A customised subclass of QTableView.

The following defaults are set:

•The default delegate is `TypedDelegate`

•The default row height is reduced to 1.5 times the text height.

•The selection model is set to `ContiguousSelection`

The following new features have been added:

•A new `currentRowChanged` signal is emitted from the view when the current row changes. In Qt, this has to be accessed through the selection model.

•A column widget may be set to the widget of a piece of text. See `setColumnWidth` for more information.

•`setModel` allows communication back to the model through the model's `setView` and `unSetView` methods. See `setModel` for more information.

•`Application.saveState` and `Application.restoreState` are supported through the `saveState` and `restoreState` methods. Currently, only the column widths are saved.

- Once a cell has been edited, the current cell moves down a row, mimicking the behaviour of most spreadsheet programs.
- The view has `copy` and `paste` support.

copyRole
The `ItemDataRole` to be used when copying (default `Qt.DisplayRole`).

pasteRole
The `ItemDataRole` to be used when pasting (default `Qt.EditRole`).

currentRowChanged(*oldIndex*, *newIndex*)
Signal emitted whenever the current row changes.

copy()
Copy the selection to the system clipboard.

Data is copied as text in a tab-separated format similar to that used by most spreadsheet programs.

nextCell()
Move to the next cell down if possible.

If the current cell is in the last row, nothing happens.

paste([*text*])
Paste tabular data into the table, overwriting existing.

The data is written to each cell using `model().setData` with the role specified in `pasteRole`. Note that no data conversion is done and all the data is pasted as strings.

The exact operation of this depends on the selection: If a range of cells is selected, then `pasteToSelection` is used. If nothing is selected, `pasteAll` is used.

If *text* is not specified, the contents of the clipboard are used.

pasteAll(*data*)
Paste data to the current index, filling down and right.

data is an iterable of rows, each row being an iterable of columns. As much of the data is pasted as possible, filling down and right from the current index. Pasting stops either when rows and columns run out or when the data runs out. The data is pasted one row at a time and a check is made after each row to determine if there is space for more.

pasteToSelection(*data*)
Paste data to overwrite the selected indexes.

data is an iterable of rows, each row being an iterable of columns. The data is pasted to fill the selected range, repeating as necessary. The selection is assumed to be contiguous between the smallest and largest selected indexes.

restoreState(*state*)
Restore the object state to *state*.

saveState()
Return the state of the object to save using `Application.saveState`.

setColumnWidth(*column*, *width*)
Sets the width of the given *column* to the *width* specified.

width may be a string, in which case `textWidth` is used to calculate the width of the text on this widget.

setModel(*model*)
Set the model displayed in the view.

When a model is set, its `setView` method is called if it exists. Similarly, `unsetView` is called when the model is removed from the view. These methods allow the model to perform specific action on the view, e.g. connecting to some of its signals.

setModels (**models*)
Connect a list of proxy models to this view.

This is a convenience function for common cases when a `DataModel` with a series of proxy models are used. The models listed should be in hierarchal order, e.g.:

```
setModels(proxy1, proxy2, datamodel)
```

is equivalent to:

```
setModel(proxy1)
proxy1.setSourceModel(proxy2)
proxy2.setSourceModel(datamodel)
```

The last model should be an actual data model, not a proxy.

3.5 Core Tools

qte.**compactSeparators** (*toolbar*)
Remove adjacent separators in a `QTooBar` instance.

qte.**error** (*prompt*)
Use a `QMessageBox` to display an error message.

qte.**fadeBrush** (*colour*[, *fraction=0.0*])
Return a brush with *colour* faded by *fraction*.

fraction should be between 0 and 1, where 0 has no effect on the colour and 1 results in white.

qte.**float2** (*s*[, *default=0.0*])
Convert *s* to a float, returning *default* if it cannot be converted.

```
>>> float2('33.4', 42.5)
33.4
>>> float2('cannot convert this', 42.5)
42.5
>>> float2(None, 0)
0
>>> print(float2('default does not have to be a float', None))
None
```

qte.**getOpenFileName** (*filters[, title][, path]*)
Display an "Open File" dialog.

Parameters

- **filters** – A list of file filters to apply.
- **title** – The dialog title. The default is `'Open'`.
- **path** – The path at which to open the dialog.

Returns The name of the file, or `None` if the dialog is canceled.

qte.**getOption** (*prompt*, *options*[, *current=0*])
Display a dropdown list of options.

Parameters

- **prompt** – Prompt text to display.
- **options** – A list of strings.
- **current** – The initially selected index.

Returns The newly selected index

qte.**getSaveFileName**(*filters[, title][, path]*)
Display a "Save File" dialog.

Parameters

- **filters** – A list of file filters to apply.
- **title** – The dialog title. The default is `'Save'`.
- **path** – The path at which to open the dialog.

Returns The name of the file, or `None` if the dialog is canceled.

qte.**getText**(*prompt*[, *default=''*, *validate=None*, *handle_invalid='confirm'*])
Requests a text string through a gui prompt.

Parameters

- **prompt** – Prompt text to display.
- **default** – Default value in the entry widget.
- **validate** – A function which accepts a single argument and returns `True` or `False`.
- **handle_invalid** – Control how an invalid entry is handled.

Returns The value entered, or `None` if the request was cancelled.

handle_invalid can be any one of the following strings

Value	Description
'confirm'	A confirmation dialog is displayed, asking whether to accept.
'warn_accept'	A warning dialog is displayed and the value is accepted.
'warn_deny'	A warning dialog is displayed and the value can be re-entered.
'cancel'	Nothing is displayed or accepted.

qte.**guessWidget**(*value*)
Attempt to guess which widget to provide based on a value.

The return value is a class which implements `EditWidgetABC`. *value* is the value the widget should work with. If no suitable widget is found, `TextEdit` is returned.

qte.**int2**(*s*[, *default=0*])
Convert *s* to an int, returning *default* if it cannot be converted.

```
>>> int2('33', 42)
33
>>> int2('cannot convert this', 42)
42
>>> print(int2('default does not have to be an int', None))
None
```

qte.**menuToToolBar**(*menu*[, *toolbar*]) → toolbar:
Add all actions in a `QMenu` to a `QTooBar`.

If *toolbar* is missing a new `QToolBar` is created, otherwise *toolbar* is updated. In either case, the toolbar is returned.

Actions are added in the same order as they were to the QMenu instance. Separators are added to correspond to those in the menu, and submenus are also added to one level, surrounded by separators. Further submenus are added as dropdown menus.

qte.**message**(*prompt*)
Use a `QMessageBox` to display an information message.

qte.**question**(*prompt*)
Display a question dialog with Yes and No buttons.

The return value is `True` if *Yes* was selected or `False` if *No* was selected.

qte.**runApp**(*mainWindow*, *name*)
Create a new application and runs it, setting the main window and name.

This also sets the locale to the current system locale.

Example usage is:

```
from gui import MainWindow

if __name__== '__main':
    runApp(MainWindow, 'My Application')
```

Once the application created has been terminated, runApp calls sys.exit, so this should be the final command to be run.

mainWindow can be either a QWidget instance or subclass. If it is a subclass, then an instance is created.

This is implemented by calling the following:

```
locale.setLocal(locale.LC_ALL, '')
QXApplication()
win = mainWindow()
QXApplication.setApplicationName(name)
QXApplication.setMainWindow(win)
win.setWindowTitle(name)
win.show()
sys.exit(QXApplication.exec_())
```

class qte.**SafeWriter**(*name*[, *text*[, *backup*]])
A context manager which provides a safe environment for writing a file.

SafeWriter attempts to avoid race conditions and ensure that data is not lost if for any reason writing fails. It works by using the following procedure for writing to a file:

1. A new empty file is created for writing using tempfile.mkstemp.
2. Once the temporary file is closed, the the original is copied to a new, temporary name (or, optionally, a specified backup).
3. The new file written is renamed to the original file name.
4. If a backup name is not set, the temporary backup is deleted.

This means that there is always a copy of the original file until after the new file is closed and if there is any failure during the process the files are rolled back to their original status.

For example:

```
with SafeWriter('file') as f:
    f.write(b'spam')
```

This is all done to avoid race conditionals as far as possible, given the provisions of tempfile.mkstemp and os.rename. Under normal circumstances, the only possibility of a race condition is that a new file with the same name as the target could be created after the target is removed and before the temporary file is renamed. This will only be possible on certain platforms where os.rename does not automatically overwrite.

There are occasions when access to the temporary file name is preferred to an open file, for example, when using with sqlite. In this case, *open* can be set to False to obtain the file name:

```
with SafeWriter('file', open=False) as name:
    conn = sqlite3.connect(name)
    conn.execute('CREATE TABLE temp (col)')
    conn.close()
```

Parameters

- **name** – The name of final file (not the temporary one)

- **text** – Indicates whether the tile should be opened in text (`True`) or binary (`False`) mode (default). On some platforms this makes no difference.
- **backup** – If this is set and the target file already exists, it is copied to *backup* before it is overwritten.
- **open** – If `True` (default), open the file and return a file object. Otherwise, return the temporary file name

`qte.`**`standardBrush`**(*color_role*)
Return a brush for a Qt standard color role.

`qte.`**`standardIcon`**(*icon*)
Return a standard icon in the application style.

icon may either be a `QStyle.StandardPixmap` flag (e.g. `QStyle.SP_DirIcon`) or the name of a standard pixmap, omitting the 'SP' portion (e.g. `'DirIcon'`).

`qte.`**`textHeight`**(*widget*[, *text*])
Return the height of *text* painted by *widget*.

If *text* is omitted, then the height of a single character is returned.

`qte.`**`textWidth`**(*widget*[, *text*])
Return the width of *text* painted by *widget*.

If *text* is omitted, the average character width is returned.

Python Module Index

q