How to use pyqtgraph

There are a few suggested ways to use pyqtgraph:

- From the interactive shell (python -i, ipython, etc)
- Displaying pop-up windows from an application
- Embedding widgets in a PyQt application

Command-line use

PyQtGraph makes it very easy to visualize data from the command line. Observe:

```
import pyqtgraph as pg
pg.plot(data)   # data can be a list of values or a numpy array
```

The example above would open a window displaying a line plot of the data given. The call to `pg.plot` returns a handle to the `plot widget` that is created, allowing more data to be added to the same window. **Note:** interactive plotting from the python prompt is only available with PyQt; PySide does not run the Qt event loop while the interactive prompt is running. If you wish to use pyqtgraph interactively with PySide, see the 'console' example.

Further examples:

```
pw = pg.plot(xVals, yVals, pen='r')  # plot x vs y in red
pw.plot(xVals, yVals2, pen='b')

win = pg.GraphicsWindow()  # Automatically generates grids with multiple items
win.addPlot(data1, row=0, col=0)
win.addPlot(data2, row=0, col=1)
win.addPlot(data3, row=1, col=0, colspan=2)

pg.show(imageData)  # imageData must be a numpy array with 2 to 4 dimensions
```

We're only scratching the surface here–these functions accept many different data formats and options for customizing the appearance of your data.

Displaying windows from within an application

While I consider this approach somewhat lazy, it is often the case that 'lazy' is indistinguishable from 'highly efficient'. The approach here is simply to use the very same functions that would be used on the command line, but from within an existing application. I often use this when I simply want to get a immediate feedback about the state of data in my application without taking the time to build a user interface for it.

Embedding widgets inside PyQt applications

For the serious application developer, all of the functionality in pyqtgraph is available via widgets that can be embedded just like any other Qt widgets. Most importantly, see: `PlotWidget`, `ImageView`, `GraphicsLayoutWidget`, and `GraphicsView`. PyQtGraph's widgets can be included in Designer's ui files via the "Promote To…" functionality:

1. In Designer, create a QGraphicsView widget ("Graphics View" under the "Display Widgets" category).
2. Right-click on the QGraphicsView and select "Promote To…".
3. Under "Promoted class name", enter the class name you wish to use ("PlotWidget", "GraphicsLayoutWidget", etc).
4. Under "Header file", enter "pyqtgraph".
5. Click "Add", then click "Promote".

See the designer documentation for more information on promoting widgets. The "VideoSpeedTest" and "ScatterPlotSpeedTest" examples both demonstrate the use of .ui files that are compiled to .py modules using pyuic5 or pyside-uic. The "designerExample" example demonstrates dynamically generating python classes from .ui files (no pyuic5 / pyside-uic needed).

HiDPI Displays

PyQtGraph has a method `mkQApp` that by default sets what we have tested to be the best combination of options to support hidpi displays, when in combination with non-hidpi secondary displays. For your application, you may have instantiated `QApplication` yourself, in which case we advise setting these options *before* runing `QApplication.exec_()`.

For Qt6 bindings, this functionally "just works" without having to set any attributes.

On Versions of Qt >= 5.14 and < 6; you can get ideal behavior with the following lines:

```
os.environ["QT_ENABLE_HIGHDPI_SCALING"] = "1"
QApplication.setHighDpiScaleFactorRoundingPolicy(QtCore.Qt.HighDpiScaleFactorRoundingPolicy.PassThrough)
```

If you are on Qt >= 5.6 and < 5.14; you can get near ideal behavior with the following lines:

```
QApplication.setAttribute(QtCore.Qt.AA_EnableHighDpiScaling)
QApplication.setAttribute(QtCore.Qt.AA_UseHighDpiPixmaps)
```

With the later, ideal behavior was not achieved.

pyqtgraph.Qt.mkQApp(*name=None*) [source]

Creates new QApplication or returns current instance if existing.

Arguments:	
name	(str) Application name, passed to Qt

PyQt and PySide

PyQtGraph supports two popular python wrappers for the Qt library: PyQt and PySide. Both packages provide nearly identical APIs and functionality, but for various reasons (discussed elsewhere) you may prefer to use one package or the other. When pyqtgraph is first imported, it automatically determines which library to use by making the fillowing checks:

1. If PyQt5 is already imported, use that
2. Else, if PySide2 is already imported, use that
3. Else, if PySide6 is already imported, use that
4. Else, if PyQt6 is already imported, use that
5. Else, attempt to import PyQt5, PySide2, PySide6, PyQt6, in that order.

If you have both libraries installed on your system and you wish to force pyqtgraph to use one or the other, simply make sure it is imported before pyqtgraph:

```
import PySide2  ## this will force pyqtgraph to use PySide2 instead of PyQt5
import pyqtgraph as pg
```

Embedding PyQtGraph as a sub-package of a larger project

When writing applications or python packages that make use of pyqtgraph, it is most common to install pyqtgraph system-wide (or within a virtualenv) and simply call *import pyqtgraph* from within your application. The main benefit to this is that pyqtgraph is configured independently of your application and thus you (or your users) are free to install newer versions of pyqtgraph without changing anything in your application. This is standard practice when developing with python.

Occasionally, a specific program needs to be kept in working order for an extended amount of time after development has been completed. This is often the case for single-purpose scientific applications. If we want to ensure that the software will still work ten years later, then it is preferable to tie it to a very specific version of pyqtgraph and *avoid* importing the system-installed version, which may be much newer and potentially incompatible. This is especially true when the application requires site-specific modifications to the pyqtgraph package.

To support such a separate local installation, all internal import statements in pyqtgraph are relative. That means that pyqtgraph never refers to itself internally as 'pyqtgraph'. This allows the package to be renamed or used as a sub-package without any naming conflicts with other versions of pyqtgraph on the system.

The basic approach is to clone the repository into the appropriate location in your project. When you import pyqtgraph, be sure to use the full name to avoid importing any system-installed pyqtgraph packages. For example, imagine a simple project has the following structure:

```
my_project/
    __init__.py
    plotting.py
        """Plotting functions used by this package"""
        import pyqtgraph as pg
        def my_plot_function(*data):
            pg.plot(*data)
```

To embed a specific version of pyqtgraph, we would clone the pyqtgraph repository inside the project, with a directory name that distinguishes it from a system-wide installation:

```
my_project$ git clone https://github.com/pyqtgraph/pyqtgraph.git local_pyqtgraph
```

Then adjust the import statements accordingly:

```
my_project/
    __init__.py
    local_pyqtgraph/
    plotting.py
        """Plotting functions used by this package"""
        import local_pyqtgraph.pyqtgraph as pg  # be sure to use the local subpackage
                                                # rather than any globally-installed
                                                # version.
        def my_plot_function(*data):
            pg.plot(*data)
```

Use `git checkout pyqtgraph-x.x.x` to select a specific library version from the repository, or use `git pull` to pull pyqtgraph updates from upstream (see the git documentation for more information). If you do not plan to make use of git's versioning features, adding the option `--depth 1` to the `git clone` command retrieves only the latest version.

For projects that already use git for code control, it is also possible to include pyqtgraph as a git subtree within your own repository. The major advantage to this approach is that, in addition to being able to pull pyqtgraph updates from the upstream repository, it is also possible to commit your local pyqtgraph changes into the project repository and push those changes upstream:

```
my_project$ git remote add pyqtgraph https://github.com/pyqtgraph/pyqtgraph.git
my_project$ git fetch pyqtgraph
my_project$ git merge -s ours --allow-unrelated-histories --no-commit pyqtgraph/master
my_project$ mkdir local_pyqtgraph
my_project$ git read-tree -u --prefix=local_pyqtgraph/ pyqtgraph/master
my_project$ git commit -m "Added pyqtgraph to project repository"
```

See the `git subtree` documentation for more information.