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 pyuic4 or pyside-uic. The "designerExample" example demonstrates dynamically generating python classes from .ui files (no pyuic4 / pyside-uic needed).

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 PyQt4 is already imported, use that
2. Else, if PySide is already imported, use that
3. Else, attempt to import PyQt4
4. If that import fails, attempt to import PySide.

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 PySide  ## this will force pyqtgraph to use PySide instead of PyQt4
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.

However, it is also often the case, especially for scientific applications, that software is written for a very specific purpose and then archived. If we want to ensure that the software will still work ten years later, then it is preferrable to tie the application to a very specific version of pyqtgraph and *avoid* importing the system-installed version of pyqtgraph, which may be much newer (and potentially incompatible). This is especially the case when the application requires site-specific modifications to the pyqtgraph package which may not be present in the main releases.

PyQtGraph facilitates this usage through two mechanisms. First, all internal import statements in pyqtgraph are relative, which allows the package to be renamed or used as a sub-package without any naming conflicts with other versions of pyqtgraph on the system (that is, pyqtgraph never refers to itself internally as 'pyqtgraph'). Second, a git subtree repository is available at https://github.com/pyqtgraph/pyqtgraph-core.git that contains only the 'pyqtgraph/' subtree, allowing the code to be cloned directly as a subtree of the application which uses it.

The basic approach is to clone the repository into the appropriate location in your package. When you import pyqtgraph from within your package, 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-core repository inside the project:

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

Then adjust the import statements accordingly:

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

Use `git checkout pyqtgraph-core-x.x.x` to select a specific version of the repository, or use `git pull` to pull pyqtgraph updates from upstream (see the git documentation for more information).

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-core https://github.com/pyqtgraph/pyqtgraph-cor
my_project$ git fetch pyqtgraph-core
my_project$ git merge -s ours --no-commit pyqtgraph-core/core
my_project$ mkdir pyqtgraph
my_project$ git read-tree -u --prefix=pyqtgraph/ pyqtgraph-core/core
my_project$ git commit -m "Added pyqtgraph to project repository"
```

See the `git subtree` documentation for more information.