Bundling applications with PyQtGraph

Table of Contents

1 Introduction

1.1 Can we cut to the chase?

You want to deliver a PyQtGraph application to your end users and you are not interested in the details bundling it. In this case, jump to section 6 for a step-by-step how-to on Windows or section 7 for Mac OS X.

1.2 The Problem

Setting up your computer for writing a Python application is an easy task. First install the Python interpreter (on Windows, it takes a few minutes to do so), set up extra libraries such as PyQt, SIP, and PyQtGraph, open your text editor and you are ready for coding. With the help of IDEs (Integrated Development Environments) such as Eclipse/PyDev, you can debug your application in no time.

However, creating a standalone distribution so that users can run your application can be a challenge. The tools for doing so have a rudimentary command line interface, a plethora of options, and need quite a bit of tuning for things to work as expected. When you add third-party libraries to the mix, you add another set of headaches. These libraries consist of pure Python code, native code, or a combination of both. In some cases, the bundling tool will not properly determine the set of all the modules to include in your deliverables. It might include too much – you will need to weed out the unused modules and DLLs – or too little – you may have to explicitly import several modules, or copy them in a script.

Porting your application to another platform adds extra work. File names have different conventions, certain APIs may work on one platform but not the other, loading dynamic libraries does not work the same way on all platforms (especially with regards to directory structure, dependencies and paths), libraries could have slightly different versions (for instance, Qt versions on Windows and Mac OS X may differ) and you need to use separate tools to create a redistributable package.

1.3 What will we talk about?

In this document, we will cover bundling an application (that is, to group the application, the Python interpreter and all needed modules under one single redistributable package) using the PyQtGraph library, on both Mac OS X and Windows platforms. The generated executable (and associated files) will run on a Mac or Windows machine without requiring the end user to install Python, PyQt, SIP and any other modules. In the process, we hope this guide will be useful for bundling other libraries as well.

We will use Py2exe (Windows platform) and Py2app (Mac OS X) to address our needs. There are other solutions for bundling Python applications out there which we will not describe here. Generally though, the types of problems encountered should be similar across all such tools.

While we are at it, we will list the packages to install on the Windows platform, and for Mac OS X, we will provide an easy Python / PyQt installation script through MacPorts.

1.4 What will we not talk about?

We will not delve into software installation. On Mac OS X, no installer is required, as we can very easily create a disk image file (DMG) which allows the user to drag the application to a folder (Applications for instance). On Windows, there are many existing solutions, for example NSIS or InnoSetup, that enable you to create an installer with little effort.

Other than for the purpose of this guide, we will not talk about porting your application to multiple platforms. The example provided will be very simple.

We will not talk about how to program in Python, how to develop applications with PyQt (see River Bank Computing's website for this), or how to use PyQtGraph itself.

2 Setting Up Your Environment

Before we can get started, we have to ensure all the prerequisites are in place.

2.1 Windows Platform

You should install Python 2.7, PyQt 4.9.1, and the latest versions of numpy, scipy, PyQtGraph, and Py2exe. All these packages are straightforward to install on Windows. Please refer to the instructions provided with each of these packages for installation.

2.2 Mac OS X Platform

On the Mac, there are basically 2 ways to install Python.

One approach is to download the required components, compile on the command line and hope for the best. This can be a very frustrating experience, trying to match the correct versions of Qt and SIP in particular, sometimes getting strange errors (I spent hours). According to the forums, getting NumPy and SciPy to build is tricky (I haven't even tried that).

Another approach, which we will describe here, uses MacPorts. MacPorts is a package manager taking care of downloads, dependencies, and setting up environment variables. There are hundreds of packages available for MacPorts, making installation straightforward. Developers who created these packages have already figured out which include files, modules and libraries to use, the proper compilation flags, and other intricacies. If a MacPorts package needs another one, it will automatically install it.

2.2.1 Installation with MacPorts

Disclaimer: the author has successfully performed a MacPorts installation under Mac OS X 10.7.3. If you are running 10.6, your mileage may vary. Mountain Lion (OS X 10.8) is right around the corner and may cause incompatibilities with Python or Qt.

Note: the commands below will replace your existing python (which came standard with Mac OS X) with one installed in /opt/local/bin. You can always switch back to the original python later on if you desire, but if you do so, keep in mind that you will not be able to readily use the PyQt, numpy, scipy and PyQtGraph packages (unless you have figured out how to build and install PyQt and numpy/scipy in particular… good luck with that). So, I recommend you stick to the python installed in /opt/local/bin after this installation.

One more note: you may already have installed a few python packages in the standard python distribution. In this case you can copy these packages to the new location for site-packages (see below), or you can run the install scripts for these packages again (typically, `python setup.py install`) after the MacPorts version of Python has been installed.

First, you need to install MacPorts. Installation is pretty easy. MacPorts provides a .dmg file which you can download (see https://distfiles.macports.org/MacPorts/), or you can also download a .pkg (package manager) installer.

Run the following commands from a terminal window:

```
sudo port install python27
sudo port select --set python python27
which python (you should get /opt/local/bin/python)
sudo port install py27-pyqt4
sudo port install py27-py2app
sudo port install py27-numpy
sudo port install py27-scipy
```

Listing : MacPorts installation of Python, PyQt, py2app, numpy and scipy

If all goes well, python installation through MacPorts should take about half an hour or more, depending on the speed of your machine and your network connection.

Then download PyQtGraph, extract it, and copy the pyqtgraph folder to:

```
/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages
```

Listing : Location of site-packages for MacPorts Python

3 The Simple Application

Let's start with a very simple graphing application. Here is the code.

```
import sys
from PyQt4 import QtGui
import pyqtgraph as pg

app = QtGui.QApplication(sys.argv)
pg.plot(x = [0, 1, 2, 4], y = [4, 5, 9, 6])

status = app.exec_()
sys.exit(status)
```

Listing : A very simple graphic application.

As you can see, we instantiate a Qt application object (always required), then create a plot window with a few data points, run the main message loop, and finally, we terminate and return the error code. If you run this sample from the Python interpreter, you should see something like the following screenshot:

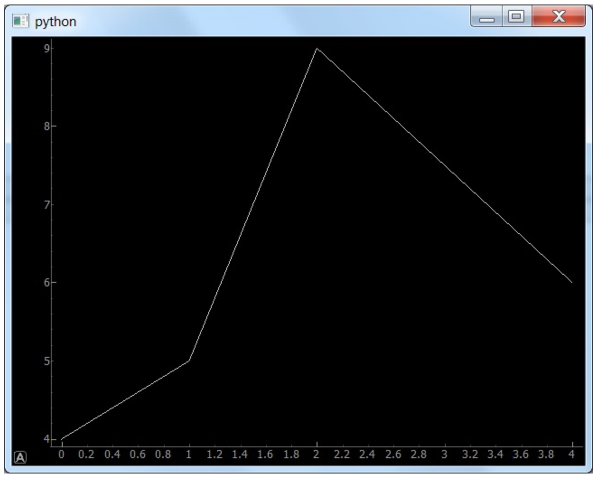


Figure : Simple graphing application, running on Windows 7.

Great. So far, so good.

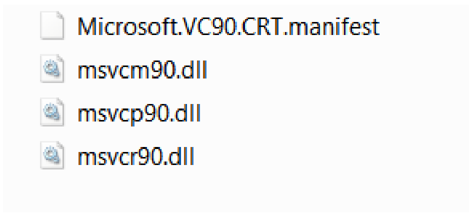
4 Bundling on Windows

Now, as we are getting quite excited redistributing our flashy graphing application to end users, we continue on a roll and put the pieces in place for bundling. The following procedure has been verified to work on Windows 7. However, if you are running Windows Vista, I do not anticipate any issue.

As mentioned before, you need to install Py2exe.

4.1 The C Runtime DLLs

The executable created by Py2exe for Python 2.7 relies on the Microsoft C runtime DLLs, the ones that come with Visual Studio 2008. You will need to find the following files and place them in a folder inside your project (in our example, redist\x86):



```
Microsoft.VC90.CRT.manifest
msvcm90.dll
msvcp90.dll
msvcr90.dll
```

Figure : Microsoft C runtime DLLs version 9.0.

You can find the C runtime here: http://www.microsoft.com/en-us/download/details.aspx?id=5582

Before redistributing this package you might want to check with your legal department though. In this guide we have included this package as a private assembly. You could also require the end users to download and install the C runtime themselves.

4.2 The Py2exe Script

Py2exe requires you to create a script telling it what to package, which additional modules and files to include, and so on.

Let's write a simple script bundling our simpleApp.pyw file. We'll name our script setupWindows.py. The reason we are adding "Windows" to the script name is that we will have a slightly different script for Mac OS X.

```
from distutils.core import setup
from glob import glob
import py2exe
import sys

sys.path.append(r'D:\xxxxxxx\Sandbox\src\redist\x86')
data_files = [("Microsoft.VC90.CRT", glob(r'D:\xxxxxxx\Sandbox\src\redist\x86\*.*'))]

setup(
	data_files=data_files,
	windows=['simpleApp.pyw']
)
```

Listing : Py2exe script for Windows.

Notice the path to the D:\ drive. Replace D:\xxxxxxx\... with the real path to your project on your system. You may want to use the current working directory instead of hard coding the path. In the paths shown above, your project code, as well as the setupWindows.py script and the batch file which we will show shortly, would be located in `Sandbox\src`.

4.3 The Build Script

As human beings, we are born lazy. So, instead of running the same command lines over and over again, we'll include them in a batch file, which we will name `buildStandAlone.bat`:

```
rmdir /S /Q dist
rmdir /S /Q build
python .\setupWindows.py py2exe --includes sip
copy redist\auto.png dist
rmdir /S /Q dist\tcl
del dist\tk85.dll
del dist\tcl85.dll
pause
```

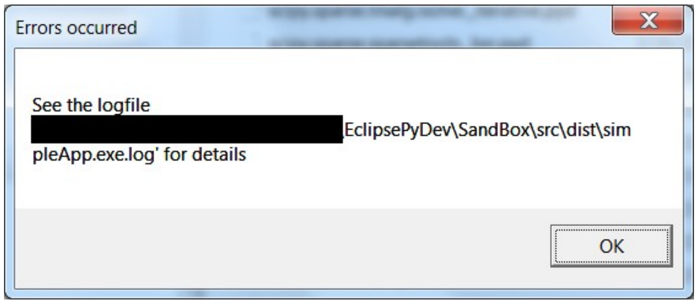
Listing : Build script for py2exe on Windows.

First, we clean up the `dist` and `build` directories, which py2exe creates. Then, we invoke Python with the Py2exe utility. After that, we copy a .png file to the redistributable directory (we will explain later why). As a last step, we remove unneeded files of tcl and tk, which we are not using. Last, we include a pause statement so that we can see what is happening (especially useful in case errors pop up).

Let's run this script! If all is well, you will end up with a "dist" directory containing our executable, "simpleApp.exe", along with the Python DLLs, the C run time DLLs, PyQt DLLs, a library.zip archive, and other .pyd files (python extension DLLs). The "build" directory is temporary and can be safely deleted (which is what our script does). For details on the directory structure, please refer to Py2exe's documentation. You might want to redirect the batch file output to a file for easier debugging.

4.4 The First Run

We cannot wait double-clicking on simpleApp.exe. Uh oh… What gives?


Errors occurred

See the logfile [redacted]EclipsePyDev\SandBox\src\dist\simpleApp.exe.log' for details

Figure : First run on Windows.

Clearly, something went bad. Let's take a look at the log.

```
Traceback (most recent call last):

  File "simpleApp.pyw", line 6, in <module>
  File "pyqtgraph\__init__.pyc", line 107, in <module>
  File "pyqtgraph\__init__.pyc", line 89, in importAll

WindowsError: [Error 3] The system cannot find the path specified:
'D:\\xxxxxx\\SandBox\\src\\dist\\library.zip\\pyqtgraph\\graphicsItems/*.*'
```

Listing : Errors running bundled executable on Windows for the first time.

Fortunately for us, Python is verbose enough when it comes to errors. Obviously here, Python is trying to look at a path which is inside a .zip archive (Py2exe creates a .zip archive where it puts all the needed Python modules).

4.5 Accessing files within the ZIP archive

If we look at pyqtgraph__init__.py at around line 89, we see the following code:

```
86      def importAll(path, excludes=()):
87          d = os.path.join(os.path.split(__file__)[0], path)
88          files = []
89          for f in os.listdir(d):
90              if os.path.isdir(os.path.join(d, f)) and f != '__pycache__':
91                  files.append(f)
92       elif f[-3:] == '.py' and f != '__init__.py':
93           files.append(f[:-3])
94      ...
```

Listing : PyQtGraph importAll function.

This code enumerates files within the given directory, strips their .py extension, and later in the importAll function, imports the modules (Py2exe sets an import hook which looks inside library.zip). It seems that we need to place a “hook” into the os.listdir( ) built in function (BIF), so that it will look inside zip files as well.

Let’s create a file called pyqtgraphBundleUtils.py for this task. Thanks to Python’s dynamic nature, it is very easy to override an existing BIF.

Here is a first implementation:

```
from zipfile import ZipFile
import os

def myListdir(pathString):
    components = os.path.normpath(pathString).split(os.sep)
    # go through the components one by one, until one of them is a zip file
    for item in enumerate(components):
        index = item[0]
        component = item[1]
        (root, ext) = os.path.splitext(component)
        if ext == ".zip":
            results = []
            zipPath = os.sep.join(components[0:index+1])
            archivePath = components[index+1:]
            zipobj = ZipFile(zipPath, "r")
            contents = zipobj.namelist()
            zipobj.close()
            for name in contents:
                # components in zip archive paths are always separated by forward slash
                nameComponents = name.split("/")
                if nameComponents[0:-1] == archivePath:
                    results.append(nameComponents[-1])
            return results
    else:
        return previousListDir(pathString)
    pass

previousListDir = os.listdir
os.listdir = myListdir

if __name__ == '__main__':
    print os.listdir("D:\\xxxxx\\src\\dist\\library.zip\\encodings")
    print os.listdir("D:\\xxxxx\\src\\dist")
```

Listing : PyQtGraph bundling utility module.

At the end of our script we have included some basic unit testing. Let’s run it. What do we get?

```
['__init__.pyc', 'aliases.pyc', 'ascii.pyc', 'base64_codec.pyc', 'big5.pyc', 'big5hkscs.pyc',
'bz2_codec.pyc', ... files omitted for brevity ... 'utf_32_le.pyc', 'utf_7.pyc', 'utf_8.pyc', 'utf_8_sig.pyc',
'uu_codec.pyc', 'zlib_codec.pyc']
['auto.png', ... files omitted for brevity ... '_tkinter.pyd']
```

Listing : Testing os.listdir( ) replacement.

In the first test, we have listed the contents of the ZIP file, and in the second test, we deferred processing to the original os.listdir BIF, which we saved into a global variable.

One thing we are noticing is the file extensions. All we have is .pyc files (Python byte compiled code), .pyd (Python extension DLLs) and others (such as .png). Since Py2exe has byte compiled everything before packaging, there will be no .py file.

The importAll function of pyqtgraph, however, looks for .py files only. With our first implementation of os.listdir, PyQtGraph will build an empty list since it didn't see any .py files, and it will fail to load any modules (silently, until we try to use PyQtGraph sub-modules like TextItem). Thus, we have to return a list of .py files only, or we need to modify PyQtGraph to also look for .pyc files. We don't want to make changes to PyQtGraph, so we will change the file extensions from .pyc to .py.

We need to change the following lines from:

```
            if nameComponents[0:-1] == archivePath:
                results.append(nameComponents[-1])
```

Listing : Appending file names to results.

to:

```
            if nameComponents[0:-1] == archivePath:
                results.append(nameComponents[-1].replace(".pyc", ".py"))
```

Listing : Renaming .pyc extension to .py.

Running our unit testing again, we obtain this time:

```
['__init__.py', 'decoder.py', 'encoder.py', 'ordered_dict.py', 'scanner.py', '_speedups.py']

… other test result omitted …
```

Listing : Testing os.listdir( ) replacement, with proper file extension.

Perfect. Let's run our batch file again (buildStandAlone.bat) but first, we have to modify our main application a bit. See how we now import our bundle utilities, before we import PyQtGraph.

```
import sys
from PyQt4 import QtGui
from pyqtgraphBundleUtils import *

import pyqtgraph as pg

app = QtGui.QApplication(sys.argv)
pg.plot(x = [0, 1, 2, 4], y = [4, 5, 9, 6])

status = app.exec_()
sys.exit(status)
```

Listing : Our main application uses the PyQtGraph bundling utility module.

For simplicity, our pyqtgraphBundleUtils.py module does not check whether we are running as a bundled application or not. It always replaces os.listdir, no matter what. This should not be an issue, as we are just deferring processing to the original built in function if we do not have to deal with ZIP files. We will leave it as an exercise to the reader to check for the presence of the ‘frozen’ attribute within the sys module (this indicates we are running inside a bundled application) and conditionally override os.listdir.

Running our bundled application again (cd dist then simpleApp.exe), we get yet another error message box. But this time, we get a different error:

```
Traceback (most recent call last):
  File "simpleApp.pyw", line 9, in <module>
  File "pyqtgraph\__init__.pyc", line 172, in plot
  File "pyqtgraph\graphicsWindows.pyc", line 55, in __init__
  File "pyqtgraph\widgets\PlotWidget.pyc", line 47, in __init__
  File "pyqtgraph\graphicsItems\PlotItem\PlotItem.pyc", line 134, in __init__
  File "pyqtgraph\graphicsItems\ButtonItem.pyc", line 15, in __init__
ZeroDivisionError: float division by zero
```

Listing : Another error while running our bundled executable.

Hmm. What happened then?

If we look at the ButtonItem.py file within pyqtgraph/graphicsItems, we can see that the constructor attempts to set an image file, then calculates a scale factor by dividing the bitmap width by the bitmap height. Clearly, the setImageFile class method must have failed, and the bitmap height was 0, which caused a divide by zero error. Looking throughout the code, we see that only the auto.png bitmap is needed.

We can pull one more trick thanks to Python’s dynamic capabilities. “Monkey patching” refers to the ability to replace a class method at run time. What we are going to do is replace the constructor of the QPixmap class, so that it will look for a file in the current working directory instead of inside the zip file. In our case, the current working directory will be “dist”, and we have already copied (through our batch file) the auto.png file that PyQtGraph needs.

As an alternate solution, we could extract the auto.png file from the library.zip file, but Py2exe does not offer the capability to add files to the library.zip archive. We could do this with a command line zip utility, but we have kept it simple for this guide.

Another solution would be to modify PyQtGraph and embed the graphic as a base64 string, but this would require changes to PyQtGraph and make it slightly harder to maintain should the graphic be updated.

Our revised pyqtgraphBundleUtils.py module looks as follows (unit testing omitted):

```
from zipfile import ZipFile
import os
from PyQt4 import QtGui

def myListdir(pathString):
    components = os.path.normpath(pathString).split(os.sep)
    # go through the components one by one, until one of them is a zip file
    for item in enumerate(components):
        index = item[0]
        component = item[1]
        (root, ext) = os.path.splitext(component)
        if ext == ".zip":
            results = []
            zipPath = os.sep.join(components[0:index+1])
            archivePath = components[index+1:]
            zipobj = ZipFile(zipPath, "r")
            contents = zipobj.namelist()
            zipobj.close()
            for name in contents:
                # components in zip archive paths are always separated by forward slash
                nameComponents = name.split("/")
                if nameComponents[0:-1] == archivePath:
                    results.append(nameComponents[-1].replace(".pyc", ".py"))
            return results
    else:
        return previousListDir(pathString)
    pass

previousListDir = os.listdir
os.listdir = myListdir

QtGui.QPixmap.oldinit = QtGui.QPixmap.__init__

def newinit(self, filename):
    # if file is not found inside the pyqtgraph package, look for it
    # in the current directory
    if not os.path.exists(filename):
        (root, tail) = os.path.split(filename)
        filename = os.path.join(os.getcwd(), tail)
    QtGui.QPixmap.oldinit(self, filename)

QtGui.QPixmap.__init__ = newinit
```

Listing : Revised PyQtGraph bundling utility module with QPixmap constructor replacement.

We set aside the previous constructor of QPixmap, and replace it with our own version. The new version will modify the filename if it is not found, and uses the current working directory.

For the last time, let's package our simple application again (run buildStandAlone.bat). Suspense… double-click on simpleApp.exe … and TADA! It now runs! Note that because we look for the graphic in the current working directory, you have to launch simpleApp.exe from within its location (either by double-clicking, or from a command prompt in its location).

4.6 Success

In this guide, we have achieved the following:

- We created a build file, so we can generate a redistributable application just by double-clicking on our batch file.

- Through an additional module, we added the ability to bundle a PyQtGraph application without requiring changes to PyQtGraph itself.

- The PyQtGraph application to bundle only needs an extra one line change, and it will work as a regular application running inside the Python interpreter, or as a bundled application created by Py2exe.

4.7 Well, almost…

We thought we were successful. We solved issues with enumerating files within a ZIP archive, we modified QPixmap a bit so that it will look for files in the working directory of our application and we required only a one line change within the application to redistribute.

Are we done? Not quite, but close. Now we want to add a text annotation to our graph. We modify the code as follows:

```
import sys
from PyQt4 import QtGui
from pyqtgraphBundleUtils import *

import pyqtgraph as pg

app = QtGui.QApplication(sys.argv)
pw = pg.plot(x = [0, 1, 2, 4], y = [4, 5, 9, 6])

text = pg.TextItem(html='<div style="text-align: center"><span style="color: #FFF;">
%s</span></div>' % "here",
                   anchor=(0.0, 0.0))
text.setPos(1.0, 5.0)
pw.addItem(text)

status = app.exec_()
sys.exit(status)
```

Listing : Adding a text item to our graphing application.

Run it from the Python interpreter and you should see the following window:

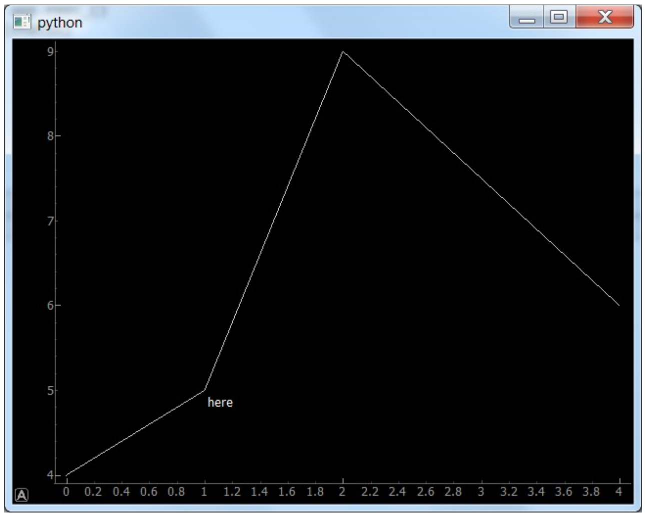


Figure : Adding a text item to our graph.

Nice.

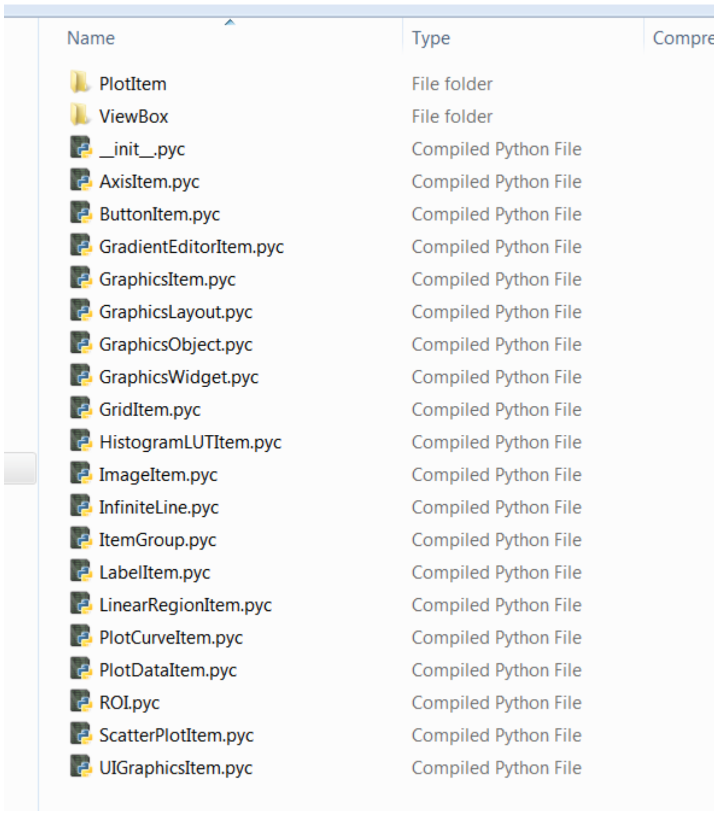
How about we bundle it again? Run buildStandAlone.bat, then launch simpleApp.exe. Woops. Now the graph appears, but an error message box pops up. If we open the log file, we can see the following:

```
Traceback (most recent call last):
  File "simpleApp.pyw", line 10, in <module>
AttributeError: 'module' object has no attribute 'TextItem'
```

Listing : Error trying to use TextItem class.

Weird. Didn't we import the PyQtGraph module already? We were able to run our application from within the interpreter, and the text item was added as expected.

Let's take a look at the library.zip in the bundled distribution:



Name	Type	Compre
PlotItem	File folder	
ViewBox	File folder	
__init__.pyc	Compiled Python File	
AxisItem.pyc	Compiled Python File	
ButtonItem.pyc	Compiled Python File	
GradientEditorItem.pyc	Compiled Python File	
GraphicsItem.pyc	Compiled Python File	
GraphicsLayout.pyc	Compiled Python File	
GraphicsObject.pyc	Compiled Python File	
GraphicsWidget.pyc	Compiled Python File	
GridItem.pyc	Compiled Python File	
HistogramLUTItem.pyc	Compiled Python File	
ImageItem.pyc	Compiled Python File	
InfiniteLine.pyc	Compiled Python File	
ItemGroup.pyc	Compiled Python File	
LabelItem.pyc	Compiled Python File	
LinearRegionItem.pyc	Compiled Python File	
PlotCurveItem.pyc	Compiled Python File	
PlotDataItem.pyc	Compiled Python File	
ROI.pyc	Compiled Python File	
ScatterPlotItem.pyc	Compiled Python File	
UIGraphicsItem.pyc	Compiled Python File	

Figure : Files in graphicsItems package.

TextItem.pyc is missing. Remember that the first error we encountered told us there was a problem trying to list all the files under the “graphicsItems” directory within the zip file.

How does Py2exe know which modules to include in the distribution? With the help of a module finder, it recursively searches all the Python source files and looks for import statements within these files. However, when the code (in this case PyQtGraph) dynamically builds a list of modules to import, the module finder cannot properly figure out the list of modules needed (unless it gets the help of a crystal ball).

There is an easy solution to this. We can explicitly import the missing module (even though it should have been dynamically imported at the time that PyQtGraph was imported):

```
from pyqtgraph.graphicsItems import TextItem
```

Listing : Explicit import of TextItem.

That way, the module finder will see our explicit import, and it will include TextItem.pyc in library.zip as expected.

If we modify our application with this solution, it becomes:

```
import sys
from PyQt4 import QtGui
from pyqtgraphBundleUtils import *

import pyqtgraph as pg
from pyqtgraph.graphicsItems import TextItem

app = QtGui.QApplication(sys.argv)
pw = pg.plot(x = [0, 1, 2, 4], y = [4, 5, 9, 6])

text = pg.TextItem(html='<div style="text-align: center"><span style="color: #FFF;">
%s</span></div>' % "here",
               anchor=(0.0, 0.0))
text.setPos(1.0, 5.0)
pw.addItem(text)

status = app.exec_()
sys.exit(status)
```

Listing : Simple application with explicit import of TextItem.

We can bundle the application again (run buildStandAlone.bat) and run it. This time, everything runs smoothly.

4.8 We got it!

Finally, we solved our library.zip path problems, patched the QPixmap constructor so that it loads for files in the application directory, and fixed another issue with a missing module.

If you use more classes from the graphicsItem package, you will need to explicitly import them so that the module finder will see them.

Additionally, the example shown was pretty straightforward. It didn't make use of OpenGL, advanced 3d graphics, etc. You may encounter additional issues that you may have to fix.

In section 6, we provide a condensed step-by-step how to for bundling PyQtGraph applications under Windows. In this final code, notice that tcl/tk libraries were removed through a py2exe option. After all, who would want to use a graphic toolkit that looks like a Motif environment from 20 years ago?

5 Now let's think different

On Mac OS X, we will encounter similar issues with bundling PyQtGraph.

5.1 The simple application

We will start from the simple application listed in section 4.7, but we will add a configuration option to disable OpenGL, which seems to have issues on Mac OS X.

```
import sys
from PyQt4 import QtGui
from pyqtgraphBundleUtils import *

import pyqtgraph as pg
from pyqtgraph.graphicsItems import TextItem
from pyqtgraph import setConfigOption

# OpenGL seems to buggy (refresh issues, crashes, etc.)
setConfigOption('useOpenGL', False)

app = QtGui.QApplication(sys.argv)
pw = pg.plot(x = [0, 1, 2, 4], y = [4, 5, 9, 6])

text = pg.TextItem(html='<div style="text-align: center"><span style="color: #FFF;">
%s</span></div>' % "here",
                   anchor=(0.0, 0.0))
text.setPos(1.0, 5.0)
pw.addItem(text)

status = app.exec_()
sys.exit(status)
```

Listing : Our simple application with OpenGL disabled.

If you followed the steps to install python, PyQt4 and PyQtGraph through MacPorts described above, you should be able to run the simple application from the Python interpreter.

5.2 The C runtime

We do not need to take extra steps to include the C runtime with our bundle. On OS X, the C runtime as available as a shared library called `libSystem.dylib`.

5.3 The Py2app script

We will create a setup script similar to the one for the Windows platform. For simplicity we have created 2 different setup scripts, one for Py2exe and one for Py2app, but we could have created a single script which checks for the platform and sets the options accordingly.

```
from setuptools import setup

APP = ['simpleApp.pyw']
DATA_FILES = []
OPTIONS = {'argv_emulation': True,
 'iconfile': '/Applications/Utilities/Grapher.app/Contents/Resources/Grapher.icns',
 'includes': ['sip', 'PyQt4']}

setup(
app=APP,
data_files=DATA_FILES,
options={'py2app': OPTIONS},
setup_requires=['py2app'],
)
```

Listing : Py2app script.

Note that on Mac OS X, you need to provide an icon for your application bundle. I just picked one from the grapher application here. We also have to explicitly list sip and PyQt4.

5.4 The build script

Next, we need a bash script that will do the packaging for us.

```
#!/bin/bash
#
# NOTE:
#
# This py2app file is designed for Python and PyQt installed through Mac Ports in /opt/local
#
rm -rf build
rm -rf dist
rm -f simpleApp.dmg
python setup.py py2app
# py2app includes too much stuff...
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtDBus.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtDeclarative.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtDesigner.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtHelp.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtMultimedia.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtNetwork.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtScript.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtScriptTools.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtSql.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtTest.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtWebKit.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtXmlPatterns.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtOpenGL.so
rm -rf dist/simpleApp.app/Contents/Frameworks/QtDBus.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtDeclarative.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtDesigner.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtHelp.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtMultimedia.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtNetwork.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtScript.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtScriptTools.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtSql.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtTest.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtWebKit.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtXml.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtXmlPatterns.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/libQtWebKit.4.dylib
# for PyQtGraph
cp /opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-
packages/pyqtgraph/graphicsItems/PlotItem/auto.png dist/simpleApp.app/Contents/Resources
# to fix bug in deployment tool
cp -R -v /opt/local/lib/Resources/qt_menu.nib/ dist/simpleApp.app/Contents/Resources/qt_menu.nib
hdiutil create simpleApp.dmg -srcfolder dist/simpleApp.app/
echo "Completed."
```

Listing : Bash script for building our application bundle.

Since we are dealing with a bash script, we need to make it executable. Open a terminal window where buildApp.sh is located, and issue the following command:

```
chmod +x buildApp.sh
```

Listing : Setting executable permission on our bash script.

Among other things, our script cleans the build and dist directories, runs py2app, removes the Qt frameworks that are not needed, copies the auto.png graphic to the Resources folder inside the app bundle, applies a fix for Qt and then creates a dmg. Phew!

Let’s run it.

```
./buildApp.sh
```

Listing : Running the bash build script.

5.5 The first run

After running the bash script, you should end up with a simpleApp.app bundle in the dist directory. Let's run it from our terminal, to ensure we will catch any errors:

```
dist/simpleApp.app/contents/MacOS/simpleApp
```

Listing : First run of our application bundle on Mac OS X.

Ouch… our icon bounced from the dock, then we got an error message that our application crashed.

Here is what we see in the command prompt:

```
Christians-Mac-mini:src Christian_Mac$ dist/simpleApp.app/Contents/MacOS/simpleApp
objc[571]: Class QCocoaColorPanelDelegate is implemented in both
/Users/Christian_Mac/Work/PythonProjects/Tools/EclipsePyDev/SandBox/src/dist/simpleApp.app/Conten
ts/MacOS/../Frameworks/libQtGui.4.dylib and /opt/local/lib/libQtGui.4.dylib. One of the two will be
used. Which one is undefined.
objc[571]: Class QMacSoundDelegate is implemented in both
/Users/Christian_Mac/Work/PythonProjects/Tools/EclipsePyDev/SandBox/src/dist/simpleApp.app/Contents/MacOS/../F
rameworks/libQtGui.4.dylib and /opt/local/lib/libQtGui.4.dylib. One of the two will be used. Which one is undefined.

... skipped many lines ...

On Mac OS X, you might be loading two sets of Qt binaries into the same process. Check that all plugins are
compiled against the right Qt binaries. Export DYLD_PRINT_LIBRARIES=1 and check that only one set of binaries
are being loaded.
QObject::moveToThread: Current thread (0x1063074a0) is not the object's thread (0x107350980).
Cannot move to target thread (0x1063074a0)

... skipped many lines ...

QWidget: Must construct a QApplication before a QPaintDevice
^CAbort trap: 6
Christians-Mac-mini:src Christian_Mac$
```

Listing : Error while running our application bundle.

In our case, Qt is trying to load libraries from 2 different locations and doesn't know which one to use. Qt suggested us to use DYLD_PRINT_LIBRARIES=1. Let's try that:

```
export DYLD_PRINT_LIBRARIES=1
```

Listing : Debugging loading/unloading of dynamic libraries.

Let's run our executable again. We will spare you the huge list of loaded libraries, but one thing attracts our attention:

```
dyld: loaded: /opt/local/lib/libpng14.14.dylib
```

Listing : Loading the libpng library.

After this message, the errors start popping. After a bit more investigating (which we will skip here), it looks like Qt is trying to load the image plugins (located in opt/local/share/qt4). Said image plugins try to load QtGui, and they do not know where to load it from. QtGui can be found inside our bundle, as well as on our system in /opt/local.

5.6 The fix

We are only using auto.png however, and png support in Qt is built in, so we need to tell Qt to stop looking for image plugins. For some reason, Qt still enumerates (or tries to enumerate) all the image plugins to query for image format support, even though it doesn't need them.

Fortunately, it is fairly easy to remove a library path from the list that Qt has when it starts up. We will append the following to our bundle utility (the full listing is given in Appendix D):

```
#----------------------------------------------------------------------------
# we need to remove local library paths from Qt
QtGui.QApplication.oldinit = QtGui.QApplication.__init__

def newAppInit(self, *args):
    QtGui.QApplication.oldinit(self, *args)
    if hasattr(sys, "frozen"):
        # on Windows, this will not hurt because the path just won't be there
        print "Removing /opt/local/share/qt4/plugins from path "
        self.removeLibraryPath("/opt/local/share/qt4/plugins")
        print "Library paths:"
        for aPath in self.libraryPaths():
            print aPath
    else:
        print "Application running within Python interpreter."

QtGui.QApplication.__init__ = newAppInit
```

Listing : Removing local library path from Qt.

We replace the QApplication constructor, check if we are running from within a bundle (the “frozen” attribute will be set in sys) and then remove /opt/local/share/qt4/plugins from the list of library paths.

Bundle the application one more time by running buildApp.sh and run it… success! We now see that the only library path left is the MacOS folder, where our executable resides. Our new QApplication constructor will work on Windows and does no harm trying to remove a non-existing library path. Besides our application bundle (simpleApp.app) our script also creates a compressed dmg file (simpleApp.dmg) which can be distributed to end users. Users can then simply drag the application to their Applications folder, or directly run it from within the dmg.

If we were to need the Qt image plugins, we would need to modify the path of their dependencies so that they will not attempt to load QtGui from /opt/local, but from the version inside our application bundle. This can be achieved with the install_name_tool command.

6 Step by Step bundling on Windows

- Install Python and other libraries as described in section 2.1
- Modify your application to import the PyQtGraph bundling utility. See Appendix A for the sample application.
- In your project directory, you need to have the following:
  - The C Runtime DLLs, in the Redist\x86 folder (see section 4.1)
  - The Py2exe Script (see Appendix B)
  - The Py2exe batch file (see Appendix C)
  - The pyqtgraphBundleUtils.py module (see Appendix D)
- Run buildStandAlone.bat to create your redistributable application.
- The dist directory contains what should be installed on the end user computer.

7 Step by Step bundling on Mac OS X

- Install Python and other libraries with MacPorts (see section 2.2.1)
- Modify your application to import the PyQtGraph bundling utility. See Appendix A for the sample application.
- In your project directory, you need to have the following:
  - The Py2app Script (see Appendix E)
  - The Py2exe shell script (see Appendix E)
  - The pyqtgraphBundleUtils.py module (see Appendix D)
- Run buildApp.sh to create your redistributable application.
- The dist directory contains an application bundle (.app) which can be run on the end user computer.
- Your project directory will also contain a disk image (.dmg) which can be opened on the end user computer.

8 Final Thoughts

8.1 Main issues

Ultimately, Py2exe and Py2app should take care of many of the issues we have seen in this guide. Like the Py2app documentation points out, the 2 most common problems are, firstly, the use of __import__ (from which the module finder cannot find which modules should be included), and secondly, the use of files within packages or modules (such as graphics or data files), which do not get included within the zip archive and that the application cannot read.

As we have found out, most of the problems we encountered are path related. A good solution would be providing hooks which would allow transparent access to regular directories or zip files. The os.listdir BIF that we replaced was a first step in this direction, but there are other path related functions (for instance, os.path.exist) which could require replacements as well.

We would still have issues with native code extensions like Qt, which make use of APIs such as fopen or CreateFile for file handling, thus bypassing regular python functions such as os.listdir(), file(), etc. A more sophisticated solution would be code injection. A DLL could get “strapped” on the back of our executable and redirect some of the API calls, such as CreateFile, to our own routines which would handle zip files. Code injection would require quite a bit of effort compared to the few hooks we added to our sample PyQtGraph application. On top of that, code injection will trigger warnings from antivirus utilities, which will either prevent your application from running, or require you to whitelist it.

Thus, replacing in an ad-hoc fashion some built in functions and a method here and there works well enough, and does not demand advanced Windows or Mac system programming expertise.

8.2 Size matters

On both Mac OS X and Windows, the generated application redistributables are quite large (about 100 MB each). This is because PyQtGraph uses numpy and scipy, and both of the latter modules import everything at once. Thus, we end up loading all the Python extension DLLs for numpy and scipy (you can verify that with the ListDlls.exe utility from Mark Russinovitch). There is not much we can trim here.

On Mac OS X with the default dmg compression, we obtain a dmg of 44 MB in size. A bit better, but still large for a simple graph application. On Windows, using InnoSetup, we can reduce the size of our setup program (setup.exe) to 15 MB (InnoSetup compresses quite well).

ActiveState has a Python recipe called “lazy importer”. It will create a module stub and only when module attributes are accessed, the module will be loaded into memory. We could envision using this, but because PyQtGraph relies heavily on numpy, we might not be able to avoid loading all the DLLs (or shared objects on Mac) in the process space on start up.

9 Appendices

9.1 Appendix A: The simple application

```
import sys
from PyQt4 import QtGui
from pyqtgraphBundleUtils import *

import pyqtgraph as pg
from pyqtgraph.graphicsItems import TextItem
from pyqtgraph import setConfigOption

# OpenGL seems to buggy (refresh issues, crashes, etc.)
setConfigOption('useOpenGL', False)

app = QtGui.QApplication(sys.argv)
pw = pg.plot(x = [0, 1, 2, 4], y = [4, 5, 9, 6])

text = pg.TextItem(html='<div style="text-align: center"><span style="color: #FFF;">
%s</span></div>' % "here",
                anchor=(0.0, 0.0))
text.setPos(1.0, 5.0)
pw.addItem(text)

status = app.exec_()
sys.exit(status)
```

9.2 Appendix B: Windows Py2exe setup script

```
from distutils.core import setup
from glob import glob
import py2exe
import sys

sys.path.append(r'D:\xxxxxx\Sandbox\src\redist\x86')
data_files = [("Microsoft.VC90.CRT", glob(r'D:\xxxxxx\Sandbox\src\redist\x86\*.*'))]

setup(
        data_files=data_files,
        windows=['simpleApp.pyw'] ,
        options={"py2exe": {"excludes":["Tkconstants", "Tkinter", "tcl"]}}
)
```

9.3 Appendix C: Windows Py2exe batch file

```
rmdir /S /Q dist
rmdir /S /Q build
python .\setupWindows.py py2exe --includes sip
copy redist\auto.png dist
pause
```

9.4 Appendix D: pyqtgraphBundleUtils.py (cross platform)

```
# pyqtgraph bundle utility
#
# Copyright (C) 2012 Christian Gavin
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included
# in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.

import os
import sys
from zipfile import ZipFile
from PyQt4 import QtGui

def myListdir(pathString):
    components = os.path.normpath(pathString).split(os.sep)
    # go through the components one by one, until one of them is a zip file
    for item in enumerate(components):
        index = item[0]
        component = item[1]
        (root, ext) = os.path.splitext(component)
        if ext == ".zip":
            results = []
            zipPath = os.sep.join(components[0:index+1])
            archivePath = components[index+1:]
            zipobj = ZipFile(zipPath, "r")
            contents = zipobj.namelist()
            zipobj.close()
            for name in contents:
                # components in zip archive paths are always separated by forward slash
                nameComponents = name.split("/")
                if nameComponents[0:-1] == archivePath:
                    results.append(nameComponents[-1].replace(".pyc", ".py"))
            return results
    else:
        return previousListDir(pathString)
    pass

previousListDir = os.listdir
os.listdir = myListdir

#-----------------------------------------------------------------------------
# look for bitmaps in current working directory
QtGui.QPixmap.oldinit = QtGui.QPixmap.__init__
```

```
def newinit(self, filename):
    # if file is not found inside the pyqtgraph package, look for it
    # in the current directory
    if not os.path.exists(filename):
        (root, tail) = os.path.split(filename)
        filename = os.path.join(os.getcwd(), tail)
    QtGui.QPixmap.oldinit(self, filename)

QtGui.QPixmap.__init__ = newinit

#---------------------------------------------------------------------------
# we need to remove local library paths from Qt
QtGui.QApplication.oldinit = QtGui.QApplication.__init__

def newAppInit(self, *args):
    QtGui.QApplication.oldinit(self, *args)
    if hasattr(sys, "frozen"):
        # on Windows, this will not hurt because the path just won't be there
        print "Removing /opt/local/share/qt4/plugins from path"
        self.removeLibraryPath("/opt/local/share/qt4/plugins")
        print "Library paths:"
        for aPath in self.libraryPaths():
            print aPath
    else:
        print "Application running within Python interpreter."

QtGui.QApplication.__init__ = newAppInit
```

9.5 Appendix E: Mac OS X Py2app setup script

```
from setuptools import setup

APP = ['simpleApp.pyw']
DATA_FILES = []
OPTIONS = {'argv_emulation': True,
 'iconfile': '/Applications/Utilities/Grapher.app/Contents/Resources/Grapher.icns',
 'includes': ['sip', 'PyQt4']}

setup(
app=APP,
data_files=DATA_FILES,
options={'py2app': OPTIONS},
setup_requires=['py2app'],
)
```

9.6 Appendix F: Mac OS X Py2app shell script

```
#!/bin/bash
#
# NOTE:
#
# This py2app file is designed for Python and PyQt installed through Mac Ports in /opt/local
#
rm -rf build
rm -rf dist
rm -f simpleApp.dmg
python setup.py py2app
# py2app is not that smart, it doesn't analyze the dependencies and includes all DLLs
# but its Windows counterpart, py2exe, does the right thing; go figure
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtDBus.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtDeclarative.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtDesigner.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtHelp.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtMultimedia.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtNetwork.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtScript.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtScriptTools.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtSql.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtTest.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtWebKit.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtXmlPatterns.so
rm dist/simpleApp.app/Contents/Resources/lib/python2.7/lib-dynload/PyQt4/QtOpenGL.so
rm -rf dist/simpleApp.app/Contents/Frameworks/QtDBus.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtDeclarative.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtDesigner.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtHelp.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtMultimedia.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtNetwork.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtScript.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtScriptTools.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtSql.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtTest.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtWebKit.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtXml.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/QtXmlPatterns.framework/
rm -rf dist/simpleApp.app/Contents/Frameworks/libQtWebKit.4.dylib
# for PyQtGraph
cp /opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-
packages/pyqtgraph/graphicsItems/PlotItem/auto.png dist/simpleApp.app/Contents/Resources
# to fix bug in deployment tool
cp -R -v /opt/local/lib/Resources/qt_menu.nib/ dist/simpleApp.app/Contents/Resources/qt_menu.nib
hdiutil create simpleApp.dmg -srcfolder dist/simpleApp.app/
echo "Completed."
```