Python3, Qt5 and PyQt5

December 19, 2014

1 Jasmine's comments

1. Developing Cross Platform Application using Qt, PyQt and PySide: Introduction - Part 1 of 5
2. Developing Cross Platform Application using Qt, PyQt and PySide: First Iteration of The Overall Application Design and Hello World! - Part 2 of 5
3. Developing Cross Platform Application using Qt, PyQt and PySide: Test Driven Development and Unit Testing - Part 3 of 5
4. Developing Cross Platform Application using Qt, PyQt and PySide: Database Support - Part 4 of 5
5. Developing Cross Platform Application using Qt, PyQt and PySide: GUI Application Development - Part 5 of 5

Install Homebrew:

```
jasminesongspro:~ jesong1126$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)
```

Qt Creator From http://www.qt.io/download/, download Qt5 download. Then

qt-opensource-mac-x64-1.6.0-7-online.dmg

will be downloaded. If you install it, Qt Creator will be installed.
Install python3, qt5 and pyqt5

```
jasminesongspro:~ jesong1126$ brew install python3 qt5 pyqt5
jasminesongspro:~ jesong1126$ brew link --overwrite python3
```

Create Gui and Py

```
jasminesongspro:~ jesong1126$ pyuic5 gs3.ui -o gs3.py
jasminesongspro:~ jesong1126$ python3 gs3.py
```

1.1 sip

SIP Reference Guide

1.1.1 Downloading

You can get the latest release of the SIP source code from
http://www.riverbankcomputing.com/software/sip/download.

1.1.2 Configuring

Unpack the source package (either a .tar.gz or a .zip file depending on your platform) and goto the unpacked sip folder.

```
$ python configure.py
```

1.1.3 Building

```
$ make
$ sudo make install
```

1.2 PyQt5

PyQt5 Reference Guide

1.2.1 Downloading PyQt5

You can get the latest release of the PyQt5 source code from
http://www.riverbankcomputing.com/software/pyqt/download5.

1.2.2 Configuring PyQt5

Unpack the source package (either a .tar.gz or a .zip file depending on your platform) and goto the unpacked PyQt5 folder.

```
$ python configure.py --qmake  /Users/jesong1126/Qt/5.3/clang_64/bin/qmake
```

1.2.3 Building

```
$ make -j 4
$ sudo make install
```

1.3 QtDesigner

You can get the latest release of the QtDesigner source code from
http://qt-project.org/downloads.

```
$ pip3 install pyqt_gs3
$ pyuic5 gs3.ui -o gs3.py
$ python3 gs3.py
$ chmod +x gs3.py
$ pip3 install bibtexvcs
$ python3 -m bibtexvcs.gui

$ pip3 install pandas
$ python3 -m pip install pandas
```

2 pythonthusiast

2.1 Developing Cross Platform Application using Qt, PyQt and PySide : Introduction - Part 1 of 5

Before we even begin this new article series, I would like to emphasize one thing : I would love to make this blog as a great starting point for you to learn many tidbits in Python software development. For example, even though what we learn is somewhat an advance topic in Python application development, I will always start with the basic. This eventually create a vast array of article series : Flask, Django, ProgrammedMe, Kivy, etc. The reason is quite simple : if you are new to a Python topic, you can always refer to the first article in a series to be able to follow the rest of the articles.

In this new article series, the target that I want to accomplished is to introduce you to the Python Android application development using either PySide or PyQt. Yes, previously you have been introduced to Kivy to develop Python application in Android. But, before going even further in Kivy, lets have a comparative experience on Python application development using another libray : PySide/PyQt. This come

with a requirement : you must be introduced to and able to develop a PySide/PyQt application in its Desktop environment. Then, we move to its Android development.This is necessary, as differ with Kivy that directly support mobile application development in its first inception, Qt support in mobile development was gradually introduced.

Great. What are you waiting for? Lets go!

2.1.1 A Gentle Introduction to Qt, PyQt and PySide

Although in this blog I haven't officially talked about cross platform GUI toolkit for Python, the truth is, I have already show you how use TkInter (which is one in many libraries in doing cross platform GUI application using Python) in this article. Needless to say, trying to elaborately explained all of the existing GUI toolkit libraries and then choosing one of them as the center of discussion/talk/tutorial, may start a discussion that lead to a flame war : all GUI toolkits have their own loyal fans. For example, as I am going to focus my discussion on Qt, Linux users may aware of its legendary rival : GTK+. Now, for those who would like to know its head to head comparison, you may headed to this WikiVs article. There you go. I just cleaned my hand.. Laughing

Qt is a C++ application framework brand and product that have its trademark being transferred from Trolltech (its original founder), to Nokia (greatly define path for Qt in mobile application development) and finally to Digia (continuing development lead, support and maintaining commercial license for Qt). Interesting thing about Qt is, even though it started as a proprietary product by Trolltech, throughout its epic development history, finally it became an Open Source LGPL product. The development of Qt is not in a closed manner inside Digia lab, but in an open governance mechanism coordinated by Qt Project. Everyone is invited to join in the Qt development!

You may wonder, "What do Python got to do with a C++ application framework?" To make the answer short : this is largely because PyQt, which is the work of of Phil Thompson starting in the late '90s, that open the door for Python developers to use Qt in their application development. It create an enormous possibility for Python developer to use Qt in much more pythonic way. Using Phil Thompson own words in foreword of the book Rapid GUI Programming with Python and Qt : "My primary goal has always been to allow Python and Qt to work together in a way that feels natural to Python programmers, while allowing them to do anything they want in Python that can be done in C++. The key to achieving this was the development of SIP". Eventually PyQt become the major product of Phil Thompson founded software consultation firm, Riverbankcomputing.

Until mid 2009, PyQt was the sole product for Python binding for Qt. This sole domination was somewhat challenged by Nokia (at that time was still owned Qt trademark) when it released PySide, which is also a Python binding for Qt. "Two product with exactly the same purpose? Why?", you may wonder. It's true. The reason is, because Nokia and the developer at Riverbankcomputing failed to reach an agreement to include LGPL in PyQt license. This means, without commercial PyQt license, you'll have to make your application that use PyQt to be also GPLed (in simple term, you must make your application source code available to the rest of the world). Up until now, Python developer who tried to use Qt in their application will be faced with two options : PyQt or PySide? The good news is, both were having (almost) the same level of compatibility in its Python level, which means Python developer can switch to use either PyQt or PySide in their application development, with little or no modifications.

2.1.2 The Edge of Changes

For those who would love to develop Python application that leverage Qt, will realize that we are somewhat standing in the bleeding edge of three changes :

1. Python itself was undergo changes from Python 2 to Python 3. Throughout its development history, Python 3 is the only release that is not backward compatible. This means application that developed using Python 2.x is not guaranteed to work in Python 3. With a large base of applications, frameworks, libraries and utilities still being developed using Python 2.x, this will somewhat make Python community hesitate to move forward to Python 3.x. You will find a lot of popular libraries simply said that it currently didn't support Python 3.

2. Starting from Qt 4.7, Qt Quick were introduced as a new framework leveraging Javascript for constructing application and its GUI. This is a competing API with the traditional (or shall I say legacy?) Qt C++ UI framework. Qt Quick leverage QML which is a descriptive language to define your application UI. QML were much like Kv language in Kivy framework. Although Qt Quick initially targeted toward mobile development, chances are it won't take long before it moves to the desktop environment. For the moment, you really have to choose one between two options in developing GUI application using Qt : Javascript QML or C++ Qt UI ?

3. Although PyQt already support Qt 5 (many thanks for this!), it dropped support for Python 2.7. Phil would like us to move forward to Python 3. What about PySide? Frankly, the current release of PySide not yet supporting Qt 5. According to PySide roadmap, "Further down the road (but not that far) is improving things at the C++ level and working on supporting qt5". Well, I believe we haven't got that support established then!

So, what's the best decision for the current state of changes in Python, Qt and PyQt/PySide? I will give a standard (and safe) answer, "It greatly depends on your application requirement or needs", or even on your boss! If however we are in the state of exploring things (like me in this blog), well, let just explore all of them and see whether interoperability, compatibility or even porting tools exists for either choices.

To make things even better, we are not restricting our exploration in Python only. We will also have a look on how it's done in its C++ counterpart. Using my personal opinion, I really love to have a holistic view for all ways, before choosing one way that will fit to the current application requirement.This will make my understanding better and thorough.

2.1.3 Installing Qt, PyQt and PySide

At the moment, I love working on Windows, with several occasion revert to Linux / Mac OS X when necessary. Therefore, for installation of this technologies, I am going to use its official download page (not using either apt, yum, brew, MacPorts, etc), listed below:

1. Download Qt 4 or Qt 5 from this download page. This is required if you like to develop Qt using C++ language. If not, you can simply skip this download. Choose the one that match your operating system.

2. Download PyQt 4 or/and PyQt 5. Downloading one of this PyQt version will include free Qt binaries (and its tools) in the appropriate version that required to run your PyQt application, which is why you don't need to download Qt separately.

3. If you want to use a great IDE for Python development in Windows, you can try to use Python Tools for Visual Studio. This is what I enjoyed to use in my current Python application development. Another solution exist of course, such as PyCharm. However, if you are about to develop Qt using C++, the Qt distribution already include QtCreator : a cross platform C++ IDE specifically designed to develop C++ Qt or Qt Quick application. With special simple configuration, it can also be use to develop non-Qt C++ application.

Downloading the above prerequisites will somewhat take a helluva time. So, grab a coffee..Or beer. I never tried beer though.. Laughing

2.1.4 What's Next?

This article should be regarded as a general introduction to Qt, PyQt and PySide. Moreover, I also laid out the current situation and condition of doing Qt development in Python using either PyQt or PySide. We will gradually increased our exploration in the next few articles starting from the development of a useful Desktop application (I am thinking of an application that fall in the category of productivity application) until the development of a Qt/PyQt/PySide mobile application (I am thinking of developing an alphabet match maker game for my kids).

By completing this article and installing the required tools, I assume that you have the interest and curiosity to develop Qt/PyQt/PySide desktop or mobile application. In that case, stay tuned for my next articles regarding this topic!

But my subsequent article will continue with the exploration of an education Kivy application though. Sorry.. Tongue out

Last comment in about 14 hours ago 14 Comments

2.2 Developing Cross Platform Application using Qt, PyQt and PySide : Introduction - Part 2-4 of 5

No need to read.

2.3 Developing Cross Platform Application using Qt, PyQt and PySide : Introduction - Part 5 of 5

This part of the series is probably the essence (and the lengthiest!) of all articles in this series: doing GUI development using Qt. Although Qt develop itself as a full stack application framework, the association of Qt is always about GUI development. That's not entirely correct, as you can completely drop GUI part of the framework, and go only with the console. But, indeed the strongest part of Qt is mainly about GUI development : starting from desktop GUI using QtGui and now targeting mobile GUI using QtQuick. In this Qt article series, we still talk about desktop GUI development using QtGui modules. Although QtQuick future may cross QtGui area by maturing itself as another option to develop desktop GUI application, but QtGui will always be there as part of Qt framework.

To add a more interesting twist on this series, we will develop all of our applications in Apple OSX operating system (still using Mountain Lion 10.8.4 though, haven't upgrade it to Mavericks). Comparing it with the official Python/PyQt/PySide distribution for Windows, if done incorrectly, preparing a working environment of all them in OSX is a lot harder. But we will have a look on how it's done correctly : with the least effort as possible.

So, what are we waiting for? Lets start our first journey toward QtGui application development!

2.3.1 General Guide on Qt GUI Application Development

As with any GUI application framework, developing GUI application using Qt GUI can be breakdown into these series of tasks:

1. Design the application user interface using provided GUI designer tool. In this regard, you will use Qt Designer to design your Qt application user interfaces, saved in file having *.ui extension. The task of using Qt Designer can be made seamless, if we develop a C++ Qt application using Qt Creator. If however we develop Qt application using Python via PyQt/PySide, we must invoke Qt Designer manually. Another point to note that, using a GUI designer is actually an optional (but highly recommended) requirement. You can develop your application GUI bare handed using plain C++/Python code. But surely, using a visual GUI designer tool to laying out your application GUI elements is much more pleasant than manually hand coding them.

2. If we are using GUI designer tool, there will be a process of importing our GUI designer file into our application code. Once again, if we are using a highly integrated IDE, this process will be seamless. You may not realize that this task ever existed, as with the case of developing C++ Qt application using Qt Creator. Developing it using Python though, will require us to invoke either PyQt pyuic4 tool or PySide pyside-uic to import our *.ui files into Python *.py code. If you have experience using Netbeans IDE to develop Java application, think of this process as the seamless process done by Netbeans when it automatically import Netbeans *.form file into Java code.

3. Connecting certain events in objects GUI element/widget event into event handler. Well, using the term events in this particular case may not fit well in Qt world. But, I put a bet that most casual GUI programmer are more familiar with the term events than the term signal and slot used in Qt. For

example, when you want to respond to a click of a button, what do you have in mind? Handling click events or connecting click signal with a slot function/method? Now, you got my point...

Qt indeed have the term events, and together with signal and slot both did have the same purpose : object interconnection. However, the way event and signal/slot was carried out are an important matter that differentiate this two concepts:

- Event is implemented as a virtual method, where as slot is a function that can exist either as an object method or as a simple function. In this regard, Qt design event to be utilize by means of object inheritance, where as signal and slot is to be implemented as a loosely coupled connection between two object.
- There are chains of events that eventually can be cut off by our overridden method, where as after emitting signal, any functions that already connected as the receiving slot can respond in parallel, without regard of the other slot.
- Events is carried out from within system events loop, making it able to receive external system events such as mouse movement or keyboard strokes. This means an event handler may took sometime to execute after event did occur. This is significantly different with signal and slot, where slot function of a connected signal is –usually– carried out immediately after a signal is emitted.

Once again, having use a good IDE can simplify your life of connecting the available signal from a widget to a user defined slot. Yes, correct, I am talking about QtCreator.

4. Defining resource file and importing it to our code. Qt has its own resource file saved in an *.qrc file which is actually a regular XML file. This is a cross platform resource file where you can define files and text resource for your application. In C++ project, the task of importing *.qrc file into *.cpp file is carried out by Qt rcc tool. This tool is called automatically by Qt Creator, where as its PyQt counterpart, pyrcc4, or its PySide counterpart, pyside-rcc, must be called manually for a Python project.

2.3.2 Preparing Qt,PyQt and PySide in OSX using Homebrew

Using Homebrew to take the role as the OSX package manager is probably the effortless way to install Python/PyQt/PySide. If Homebrew works correctly on your OSX machine, then we can say that preparing a working Python environment in OSX is way a lot easier than its Windows counterpart. It may be on a par with Ubuntu with its apt package manager. If it is as simple as doing apt-get install in Ubuntu, then it is also as simple as doing brew install on OSX. How cool is that?

Anyway, although you can inspect that OSX already shipped with Python environment (Python 2.7.2 in my Mountain Lion), but the subsequent task of upgrading/installing Python packages is not officially supported by Apple. Therefore, using Homebrew (or simply Brew) to manage our OSX packages, is simply irresistible. And it's always better to left Apple built-in python distribution to be untouched.

Now lets move on to the task of preparing Python working environment in our OSX system using brew.

1. Lets install brew using this command :

```
ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)".
```

Brew installation is very informative and a pleasant to work with. Just execute it and follow any error messages (if any), informational message or recommendation of how to make brew working as effective as possible in your OSX environment. Anyway, don't worry about that ruby thing. OSX is also shipped with ruby distribution. So, after waiting for coupla minutes, soon you will have a working brew binary ready in your OSX environment.

2. You can go ahead and install any Python and its packages using the command brew install ¡FORMULA¿. Formula is just another name for package. To search for available package, simply issue the

command brew search ¡FORMULA¿. So, to find out whether we can install qt using brew, simply issue the command brew search qt. It will give you qt, qt5, pyqt and pyqt5 amongst other things.

Recall that PyQt5 is distributed only as Python 3 package, so if you want to use PytQt4, you will end up installing Python 2.7 also. This means, eventually you will have two different version of Python. Don't worry about it though. Brew will take care of this for you.

To install Qt4, PyQt4 and PySide, issue the command brew install python qt pyqt pyside pyside-tools.

To install Qt5 and PyQt5, issue the command brew install python3 qt5 pyqt5.

Installing Qt4 or Qt5 using brew will not install QtCreator, which is an important factor for a successful C++ Qt project development. To finish the preparation you can download QtCreator manually from here. In my OSX workstation I install the full binary distribution of Qt 5.2.1 along with its QtCreator. I also install PyQt4/PyQt5/PySide along with Qt4/Qt5 using brew, to allow PyQt/PySide available in my Python 2/3 environment.

There is one last requirement should you want to develop C++ Qt application in OSX : you must do a full installation of XCode. Yep, that dreaded 1.6GB of *.dmg file must be downloaded and installed in your machine. I have try to install the XCode command line tools only, but QtCreator still need the full installation of XCode in order for it to able to build and package our C++ Qt application. The good news is you can download XCode freely once you register as Apple Developer. I purposely upgrade my Mountain Lion 10.8.2 into 10.8.4 to let me install the latest Xcode 5.0.2. That works well with the latest QtCreator shipped with Qt 5.2.1.

2.3.3 Developing Qt Widgets Application with C++ Qt

The need to produce multiple binary files

Before starting our C++ Qt project, there is one important concept to clear out first : every C++ project regardless of its framework being used or its dependent operating system, will always produce one binary file. Only with certain management, multiple binary can be produced by a C++ project. Previously we already have a Unittest project, producing one binary file namely unittest. In this project, we will build another project, a QtGui project, that will also create another binary file, lets name it pythonthusiast. This mean, special consideration must be made to allow our project produced multiple binary file. To allow a better understanding of the overall development process, I love to start with a new C++ project that produce single binary file first (the QtGui project), and then we will modify it to produce another binary file (the existing Unittest project).

If you go with the Python path, this complexity is not even known. Why? Simply because there is no concept of binary file in Python. You can feed any *.py file into Python interpreter and it will try to run it: regardless of whether this *.py file is actually your main Python file or not.

Pretty nice eh?

2.3.4 Developing Pythonthusiast as a Single C++ Qt Widgets Application

Creating a Qt Widgets Application

Lets start a new QtGui project by choosing Qt Widgets Application in the existing QtCreator template. I am going to use the name Pythonthusiast as this project name, but of course you are free to chose other name.

A Qt Widgets Application template

Accept all the default values in the subsequent wizard page, because indeed we will create a MainWindow application using QMainWindow as the base class. The important thing to note from this wizard page is that, QtCreator will present you with a Kit Selection dialog as seen below:

Kit selection dialog for shadow build feature in QtCreator

This dialog present you with the directory use by QtCreator to produce all intermediate objects and target binary(-ies) for your project. By default, QtCreator will use shadow build feature, meaning that all the resultant artifacts will be placed outside of your source code directory, which is a great feature, as we don't want our source code directory to be clutter up with buncha binary files. This configuration was saved in a *.pro.user file. So, if you move out your project to another directory, be sure to delete this *.pro.user file,

as QtCreator will create it again for you. If not, you will get this annoying warning, Qmake does not support build directories below the source directory. Another important technique is to add *.pro.user pattern in your .gitignore file.

Okay, as everything were clearly explained (I hope so), let just run our new project (hit ⌘+R in OSX or CTRL+R in other OS) and see the result. Great isn't it?

Create A New Login Dialog

Reviewing again our application flow from the first article in this series, it is clear for us that we need a login dialog. Hence, right click on our root project node and select Add New, and choose Qt Designer Form Class from the available Qt template. Choose either Dialog with Buttons Bottom or Dialog with Buttons Right, click continue and name it LoginDialog. This class is a subclass of QDialog which feature a standard popup dialog in a Qt application.

I choose the Dialog with Buttons Bottom I choose the Dialog with Buttons Bottom template

I bet you have the urge to start dragging and dropping all that cool Qt Widgets into this dialog. Well, don't be. Lets select a proper Layout first, which is in our case that would be Form Layout.

Drag and drop Form Layout to our LoginDialog form designer Drag and drop Form Layout element to our LoginDialog form designer

Place it in the proper position, and then right click inside Form Layout red rectangle guide, and then choose Add form layout row. You'll be presented with this dialog:

Add pair of QLabel and QLineEdit for each row Add pair of QLabel and QLineEdit for each row

Make two rows that comprise of –intuitively– username and password input widgets. You may realize that for the QLineEdit control I like to rename it using the pattern txt*, which is the pattern that I preserve since my Visual Basic programming career. Of course you can use any naming convention that suit you. One modification about QLineEdit for password is, you should change its echoMode to Password, which will make it behave as a standard password field. Below is the resultant login dialog:

Final user interface design of LoginDialog Final user interface design of LoginDialog

Implement Slot for LoginDialog Signals

Yeah, for veteran in other desktop GUI application framework (Delphi, VB/VC 6.0, .NET and Java) this sub chapter title may seems weird. You can mentally read it as, "Implement Event Handler for LoginDialog Events", and pragmatically, it suit just well. Right click on our Button Box that comprise of OK and Cancel button, and click Go to slot.

All the available signals for a ButtonBox widget All the available signals for a ButtonBox widget

Choose accepted() signal and press OK. QtCreator will create/direct you to the function that will act as slot for that signal. accepted() will be emitted if user press OK button in our LoginDialog. I have designed that this dialog should return three possible values that being implemented as a C++ enum as follows:

```
enum LoginStatus { Success, Failed, Rejected };
```

A built-in behaviour for QDialog subclass is to close the dialog and return QDialog::Accepted when user click OK and close the dialog and returning QDialog::Rejected when user click Cancel button. As this dialog will eventually trying to authenticate provided credential with existing one from the database, therefore I modify LoginDialog return value to suit authentication result : Success (when the credentials are a match), Failed (unmatched credentials) and Rejected (user cancel the login dialog). At this step, we haven't use our previous Auth class, therefore we simply compare txtUsername→text() to be equal with certain username.

```
void LoginDialog::on_buttonBox_accepted()
{
    if(ui->txtUsername->text()=="admin")
    {
        setResult(LoginDialog::Success);
    }else{
        QMessageBox msgBox(this);
        msgBox.setIcon(QMessageBox::Warning);
        msgBox.setWindowTitle(tr("Pythonthusiast"));
```

```
        msgBox.setText(tr("Either incorrect username and/or password. Try again!"));
        msgBox.setStandardButtons(QMessageBox::Ok);
        msgBox.exec();

        setResult(LoginDialog::Failed);
    }
}
```

Here, we use QMessageBox to display warning when authentication method is failed. Observe that we use tr() method to correctly return translated string for certain language localization. For now it simply return the given string.

Finally, connect ButtonBox rejected() signal with its slot using the following code:

```
void LoginDialog::on_buttonBox_rejected()
{
    setResult(LoginDialog::Rejected);
}
```

Using LoginDialog in Our Application

Once again, examine our application flow design, and think how we can use LoginDialog in the existing application code. For a start, our C++ Qt application is configure to execute main.cpp which contain main() function as the initial starting point for any C/C++ application. Observe the current main() function as follows:

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

The above code simply create a QApplication instance, showing MainWindow and starting application events loop. And to adjust it to suit our login feature application flow, we must use our LoginDialog class and modify this main() function as follows:

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    LoginDialog login;
    bool isAuth = false;
    int result;
    do
    {
        result = login.exec();
        isAuth = result == LoginDialog::Success || result == LoginDialog::Rejected;
    } while(!isAuth);

    if(result == LoginDialog::Success)
    {
        MainWindow w;
        w.show();
        return a.exec();
    }
```

```
    a.quit();
    return -1;
}
```

I am sure the code above is self explanatory as it really an implementation from our designed application flow. Try to run this application, and instead of directly presenting our MainWindow object, now you will be presented with a Login Dialog that either must be filled with correct credentials or cancelled. The MainWindow object will be shown only when the credentials are correct. If not, well, you'll be stuck with this LoginDialog... Laughing

You must enter proper credentials to pass this login dialog You must enter proper credentials to pass this login dialog

Modifying MainWindow

In the next article, we will work intensely in this MainWindow class. For now, we will keep ourselves satisfy by adding two menu items in it : File-¿Exit and Help-¿About. To configure menu for this MainWindow, simply open (double click) the file mainwindow.ui and start typing in the menu placeholder already provided by QtCreator. Once all the menu items were added, you can directly write the slot function that will handle triggered() signal for that particular menu item. Example given for Exit menu item as seen below:

Connecting Exit menu item triggerred() signal to slot function Connecting Exit menu item triggerred() signal to slot function

For the exit slot function itself, we will confirm user about this action by using the following code:

```
void MainWindow::on_actionExit_triggered()
{
    QMessageBox msgBox(this);
    msgBox.setIcon(QMessageBox::Question);
    msgBox.setWindowTitle(tr("Pythonthusiast"));
    msgBox.setText(tr("Are you sure you want to quit?"));
    msgBox.setStandardButtons(QMessageBox::No|QMessageBox::Yes);
    msgBox.setDefaultButton(QMessageBox::Yes);

    if(msgBox.exec()==QMessageBox::Yes)
    {
        qApp->quit();
    }
}
```

Whereas for the About menu item, we use the following slot function, that will display an informational dialog box about our application:

```
void MainWindow::on_actionAbout_triggered()
{
    QMessageBox msgBox(this);
    msgBox.setIconPixmap(QPixmap(":/images/sherlock.png"));
    msgBox.setWindowTitle(tr("Pythonthusiast"));
    msgBox.setText(tr("Well Watson, isn't it obvious to you that Qt rocks?"));
    msgBox.setStandardButtons(QMessageBox::Ok);
    msgBox.exec();
}
```

Notice something different there? Instead of the default icon provided by QMessageBox, we use our own icon in it. This feature is provided by means of Qt Resource file (*.qrc), which is simply an XML file that will be processed by Qt rcc tool (automatically if you are using QtCreator) into a *.cpp code.

To manage your resource file, add a Qt Resource File to your project, name it app.qrc (or anything that you like) and you will be able to use QtCreator easy to use resource editor to manage your *.qrc file.

Qt easy to use resource editor Qt easy to use resource editor

First, you have to prepare all your resource files to be stored inside of your source directory. Of course you can use sub directory to better manage its layout. Second, you will have to add a prefix for your resource files (e.g /images, /icon, /art, etc.). This is not to be confused with where you put your actual resource files. After adding a prefix, click it and then start to add file for that particular resource prefixes. And... voila! Your resource files are ready to use like the example given above.

Conclusion

This conclude a detailed explanation of how to create a Qt Widgets Application, that produce a single binary. You can find the resultant binary file in your shadow build directory. But this project missing one important feature : how can we use our previous Auth class and helper file in our previous Unittest project?

Ready to move on? Great. Lets continue our journey on the subsequent section talking about a subdirs project template.

2.3.5 Developing Pythonthusiast as a Multiple C++ Qt Project

Introducing Subdirs Project Template

In earlier section, I've introduced you to the reason why we need a C++ Qt project that produce multiple binary file. And now, we will have a look on how it is done. Qmake has as special project template called subdirs template. Project created with subdirs template will only serve as the container for any other Qmake projects (called subprojects). Contrary to its name that may lead you to think that these subprojects must reside in a sub directory of the Subdirs project, well, in practice you can put it anywhere. Even in the same directory! (Now you can literally say goodbye to the lexical meaning of subdirs..) As you know that every Qmake project is defined in a *.pro file, this means with a subdirs project basically you are only managing these multiple *.pro files (called subprojects) using a single *.pro file (called subdirs project), that lets you create multiple binary for each of the subprojects.

Creating Subdirs Project

To create a subdirs project, choose Other Project-¿Subdirs Project from the available project template.

Subdirs Project of QtCreator Subdirs Project of QtCreator

In the final page of the Subdirs Project you'll be asked to start Adding Subprojects. Click this button. But in doing so, QtCreator will prompt you to create a new Qt project that will reside in subdirectory of this Subdirs project (Yeah, the meaning of subdirs project is preserve this way). As we already have our existing Qt Widgets project from earlier section and an existing Unittest project from earlier article, we will have to dismiss this add new project dialog. Click Cancel, and QtCreator will return you to the editor of your new subdirs project *.pro file. Here we will add our own *.pro files, using these configuration:

```
TEMPLATE = subdirs

SUBDIRS += Pythonthusiast \
           Unittest

Pythonthusiast.file = src/Pythonthusiast.pro
Unittest.file = test/Unittest.pro
```

In my project configuration, I create two subdirectory : src and test. All of our application source code will be placed in src directory, while all the test related source code will be placed in test directory. Before saving this new subdirs *.pro file, you will have to copy all of our previous Pythonthusiast (or whatever you name your Qt Widgets application earlier) to this src directory. For the Unittest project, except for the Unittest.pro file and TestCase01.cpp that goes to test directory, all other source code (auth.* and helper.*) are also goes to src directory. This is because we are going to use these two files as part of Pythonthusiast project. We will need to really authenticate our users, right?

Save it, and QtCreator will automatically list all of subprojects defined in this *.pro file. Pretty neat, eh?

Subdirs Project with Two Subprojects Subdirs Project with Two Subprojects

With this configuration, you can automatically build all of Subprojects binary easily, or you can also build and run each of subprojects manually. The usual case is that, you will frequently work with core features of the application and test them by running Unittest project.

Finally, as we modify our Unittest project files location, we have to reflect this changes in the existing Unittest.pro file, using these configuration:

```
QT += testlib sql
TEMPLATE = app
TARGET = Unittest
DEPENDPATH += .
INCLUDEPATH += .
CONFIG   -= app_bundle

# Input
HEADERS += \
    ../src/helper.h \
    ../src/auth.h
SOURCES += TestCase01.cpp \
    ../src/helper.cpp \
    ../src/auth.cpp
```

In the above configuration, we point HEADERS and SOURCES to the correct location of our auth.* and helper.* file. The similar modifications (adding auth.* and helper.*) must also be made into Pythonthusiast.pro file as follows:

```
SOURCES += main.cpp\
        mainwindow.cpp \
    logindialog.cpp \
    auth.cpp \
    helper.cpp

HEADERS  += mainwindow.h \
    logindialog.h \
    auth.h \
    helper.h
```

Modifying Pythonthusiast Project to Use The Existing Authentication Feature

As we have already made auth.* and helper.* available in our Pythonthusiast project, now we can easily use them (and feel safe because it is already test-proof). Our task will come into two part : Connecting to the database We simply call Helper::dbConnect() in main.cpp to open our database connection. Beware that, in certain application this behaviour may not appropriate. For example, what if our application connect to a networked PostgreSQL database and the network is not available? Our application will behave not as expected for sure and confuse user. But, as we use a local and created on the fly SQLite database, this will not pose this kind of issue. The database will always be available. Authenticate user credentials We simply replace the logic of checking the entered user credentials by using our test-proof Auth::doLogin method

```
if(auth.doLogin(ui->txtUsername->text(), ui->txtPassword->text()))
            { ... }
```

Done.

We have create a solid and fully functional (but not yet complete) C++ Qt MainWindow application. In the next section, we will have a hands-on on how to create the same application in PyQt/PySide. Almost

all of the C++ code can be directly translated to the Python version, but without the aid of the same level comfortable develoment feature that QtCreator provided.

2.3.6 Developing Qt Widgets Application with PyQt

General Guide on Developing Qt Application Using Python

Foreword warning : so far, there is no Python IDE equivalent in Qt development features compare to QtCreator. You will have to go back and forth between your Python IDE and QtDesigner to design application GUI interfaces, manually issue all counterpart command line utility of uic (import QtDesigner *.ui file into Python *.py code) and rcc (import Qt resource file *.qrc into Python *.py file) in its PyQt/PySide version and connect manually all your signal widgets to slot function. It lead me to think that, if there is no strict Python requirement in your Qt project, just use C++. Live can be so much easier. Or, in another perspective, imagine if someday they make Python support directly accessible in QtCreator. Now that's a simple life...

Anyway, in this PyQt project (and the later PySide version) we are going to use PyCharm Community Edition for Mac OS X as our Python IDE. It's a great IDE, despite its lack of tight support in PyQt/PySide application development. But then again, there is no Python IDE having that kind of support... So, PyCharm CE suit us just well...

Creating A PyQt Project in PyCharm

Well, it's actually a regular Python project. Nothing special. Click File-¿New Project and choose your appropriate Python interpreter (don't worry adding one can be trivially done using PyCharm, just click on that little ellipsis button on the right of Python interpreter combo box)

Create a new Python project Create a new Python project

Okay, what do you get? Just a plain Python project with nothing in it.. Laughing To mimic our C++ Qt project, right click on the root Project node (or simply click File-¿New) and choose Python file. Gives it the name main. As previously we already finished our main.cpp, here we will simply translate its into Python code. It really is straightforward. Have a try on this process!

```
import sys
from PyQt4.QtGui import QApplication
from logindialog import LoginDialog
from mainwindow import MainWindow
import helper

if __name__ == "__main__":
    a = QApplication(sys.argv)
    helper.dbConnect()

    loginDialog = LoginDialog()

    isAuth = False
    result = -1
    while not isAuth:
        result = loginDialog.exec_()
        if result == LoginDialog.Success or result == LoginDialog.Rejected:
            isAuth = True
        else:
            isAuth = False


    if result == LoginDialog.Success:
        w = MainWindow()
        w.show()
        a.exec_()
```

```
    sys.exit(-1)
```

Surely Python will complains a lot if you try to run this main.py file (CTRL+SHIFT+R). Beside copying all previous *.py file from our Unittest project (auth.py, helper.py and test_features01.py) to this project directory, we will also need to prepare mainwindow.py and logindialog.py which are classes derived from generated class from its respective *.ui files. Lets see how it is done using PyQt.

Developing LoginDialog Interface

If previously we can easily create Qt Designer Form Class from within QtCreator itself, now the case is different with PyQt. You will have to manually run QtDesigner (hit ? + SPACE and type Designer), create new Qt Designer Form, design it as you would in QtCreator and finally save it in your PyQt project directory. After that, you will have to open a terminal, and manually issue this command : pyuic4 logindialog.ui -o ui_logindialog.py. Have a look at this automatically generated Python file. It contains UI_LoginDialog class that will construct our LoginDialog GUI. I use the pattern ui_* in the generated Python file, as it is also the pattern use by QtCreator when generating *.cpp from *.ui files (have a look at your C++ Qt project shadow build folder!).

Finally, (yes, sorry, there is still one final task must be done), you will have to subclass this UI_LoginDialog class in a new class named LoginDialog and saved it in a file named logindialog.py. It is in this class, that you finally develop your LoginDialog behaviour. Inspect this LoginDialog code snippet:

```
class LoginDialog(QDialog, Ui_LoginDialog):
    Success,Failed,Rejected = range(0,3)
    def __init__(self):
        QDialog.__init__(self)
        self.setupUi(self)
        QtCore.QObject.connect(self.buttonBox, QtCore.SIGNAL(_fromUtf8("accepted()")), self.onAccept)
        QtCore.QObject.connect(self.buttonBox, QtCore.SIGNAL(_fromUtf8("rejected()")), self.onReject)

    def onAccept(self):
        auth = Auth()
        if auth.doLogin(str(self.txtUsername.text()), str(self.txtPassword.text())):
            self.setResult(self.Success)
        else:
            msgBox = QMessageBox(self)
            msgBox.setIcon(QMessageBox.Warning)
            msgBox.setWindowTitle(_translate("LoginDialog", "Pythonthusiast", None))
            msgBox.setText(_translate("LoginDialog", "Either incorrect username and/or password. Try ag
            msgBox.setStandardButtons(QMessageBox.Ok)
            msgBox.exec_()
            self.setResult(self.Failed)

    def onReject(self):
        self.setResult(self.Rejected)
```

Observe that the above class is derived from Qt QDialog class and the previously generated UI_LoginDialog class. In the constructor of this class, we connect particular widgets signal to Python methods that serve as its slot function. Until this point you may already realize how comfortable it is to let QtCreator do these code plumbing for you. Yeah, it'd be wonderful if QtCreator also support Python development in the future..

An important note about Qt *.ui files are, it's a cross platform and language independent code. Meaning, the existing *.ui files that were created for a C++ Qt project, can be reused in another Qt project : either PyQt/PySide. So, if you are following this tutorial as it is, you can copy all your existing *.ui files from the previous C++ Qt project to the PyQt project directory and directly generate *.py files using pyuic4 tool.

Great. Now that you have completed our LoginDialog, you can try running the application again. If all goes well, you'll have the same Qt application as previously developed using C++.

Developing MainWindow Interface

After finishing our LoginDialog, the next step is to polish our MainWindow by adding (as previously done with the C++ version) File-¿Exit and Help-¿About. Now that you have the knowledge to generate *.py from *.ui files using pyuic4 tool, I can directly show you the code for MainWindow class.

```
from PyQt4 import QtCore, QtGui
from PyQt4.QtGui import QMainWindow, QMessageBox, QPixmap
from PyQt4.QtGui import QApplication
from ui_mainwindow import Ui_MainWindow

class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.setupUi(self)
        self.actionE_xit.triggered.connect(self.onExit)
        self.action_About.triggered.connect(self.onAbout)

    def onExit(self):
        msgBox = QMessageBox(self)
        msgBox.setIcon(QMessageBox.Question)
        msgBox.setWindowTitle(_translate("MainWindow", "Pythonthusiast", None));
        msgBox.setText(_translate("MainWindow", "Are you sure you want to quit?", None))
        msgBox.setStandardButtons(QMessageBox.No|QMessageBox.Yes)
        msgBox.setDefaultButton(QMessageBox.Yes)
        msgBox.exec_
        if msgBox.exec_() == QMessageBox.Yes:
          QtGui.qApp.quit()

    def onAbout(self):
        msgBox = QMessageBox(self)
        msgBox.setIconPixmap(QPixmap(":/images/sherlock.png"))
        msgBox.setWindowTitle(_translate("MainWindow", "Pythonthusiast", None))
        msgBox.setText(_translate("MainWindow", "Well Watson, isn't it obvious to you that Qt rocks?", N
        msgBox.setStandardButtons(QMessageBox.Ok)
        msgBox.exec_()
```

Now, what about our app.qrc ? How can we create such file in our PyCharm? Well, as it was just an *.xml file, you can easily create a new empty file, name it into app.qrc, and manually hand code its content by looking at app.qrc previously created by QtCreator. For your reference, this is the generated by QtCreator (well, actually it was created by QtDesigner embedded inside QtCreator):

```
<RCC>
    <qresource prefix="/images">
        <file>sherlock.png</file>
    </qresource>
</RCC>
```

Pretty easy isn't it? But QtDesigner can create this file for you using one of its pallet : Resource Browser. Try to create a new app.qrc file or use existing one created form the previous C++ Qt project. But still, for each changes in the resource file, you'll have to manually issue the command : pyrcc4 app.qrc -o app_rc.py. Why the name app_rc.py? Because it's the default name use by QtDesigner when using imported resource file for your GUI in *.ui files. Failed to do so, Python will warn you about this missing file, so you can generate it manually.

Great. We have finish our Pythonthusiast PyQt application and now we are ready to move on to convert this PyQt application to use PySide. Lets see whether we really don't have to do any other extra works than simply to replace all import of PyQt into PySide.

2.3.7 Developing Qt Widgets Application with PySide

Change All PyQt Import Into PySide

An important thing to note is, you will eventually ended up using PyQt or PySide. Certainly not both. The first task in converting a PyQt application into PySide is removing all import of PyQt and then change it into PySide. The good news is, using a great Python IDE such as PyCharm will greatly ease us on this process (you don't have to remember PySide equivalent of import statement for particular PyQt import). Simply delete all import statement, wait for a while, go to the first line of Unresolved references problem in your code and then hit CTRL+ENTER. PyCharm will show you the following popup dialog:

Automatically fixes all unresolved import

Choose Fix all 'Unresolved references' problems and .. voila! Your PyQt code is safely turn into PySide. Nice, eh?

2.3.8 Use PySide Tools instead of PyQt

Recall that we need certain PyQt tools to make our Qt specific files (*.ui and *.qrc) available from our Python code. Using PySide certainly will require us to use its own tools. So, instead of using pyuic4 to generate *.py from *.ui you, will have to use pyside-uic. And instead of using pyrcc4 to generate *.py from *.qrc, you will have to use pyside-rcc. The same usual workflow apply here. Even the command line parameters are the same!

And that's all that you need to use PySide instead of PyQt...

2.3.9 Prepare For Differences Between PyQt and PySide

Understanding API Level 1 and Level 2 in PyQt

The current PyQt4, was shipped with PyQt API Level 1 and API Level 2. For Python 2.x, it's defaulted to API Level 1. PySide itself is only supporting PyQt API Level 2. This mean, to port PyQt application to PySide as effortless as possible is to use an API Level 2 of PyQt. The important feature in API Level 2 is, there is no Qt QString, QVariant, etc. They automatically converted into Python relevant data types.

An excellent example is our Helper::computeHash() method, shown below:

```
QString Helper::computeHash(QString original)
{
    return QCryptographicHash::hash(original.toUtf8(),QCryptographicHash::Md5).toHex();
}
```

To use a Qt centric code, we use this Python code in PyQt:

```
def computeHash(original):
    return QCryptographicHash.hash(QString(original).toUtf8(), QCryptographicHash.Md5).toHex()
```

It really is a 1-on-1 translation from its Qt version, right? But, as you may already know, this Python code wont' run in PySide. It will complained when you do the login process, "argument 1 has unexpected type of 'QString'". To use a much more Pythonic way, lets convert the above method into this one:

```
def computeHash(original):
    return QCryptographicHash.hash(original, QCryptographicHash.Md5).toHex()
```

No QString there, and it works just great in PySide. In PyQt? You have to first set its API Level to 2, and it also works flawlessly.

```
import sip
sip.setapi("QString", 2)
```

This code must come first in the file main.py –or any Python file that start your application. Failed to do so, Python will raised error : ValueError: API 'QString' has already been set to version 1.

PySide Failed to Show Non-Native Menubar on OSX

In this project, I encounter an issue with PySide on OSX where its application menubar won't showing in the application MainWindow, even though I have already set menuBar nativeMenuBar to False. Eventually I found out that the problem lies in the glitch, that PySide still use self.centralWidget = QtGui.QWidget() in ui_mainwindow.py even though nativeMenuBar set to False, where it should be self.centralWidget = QtGui.QWidget(MainWindow) to let the menu bar live inside of application's MainWindow. You got to manually fix this glitch, after each running of pyside-uic. This case is not happening in PyQt.

This issue arise because we are developing an application targeting Apple OSX, where it has strict User Interface Guidelines regarding application menu bar to be moved out to OSX unified system menu bar, instead of living in the application MainWindow. However, in this section I opted to still use non-native menu bar, and move the discussion of making an application that conform to user's Operating System to the last article of this series where we will discuss in great length about how to package a C++ Qt/PyQt/PySide application in Windows/OSX/Linux.

2.3.10 Conclusion

Phew... This is probably the lengthiest article I've ever written in this site! And if you do read this article until this conclusion section, well, I am pretty sure you have serious problem with enthusiasm in Qt Laughing

There are two more articles that I would like to write in this series : Part 6 about developing this Pythonthusiast application as a Qt application that talk to a back-end Django API (that get designed and built with Apiary.io) and Part 7 about packaging these Qt application into its native independent distribution format (using py2exe for Windows and py2app for OSX). But, as I think those two articles can be viewed as an independent articles, hereby I wrap up these series into five parts, making this article as the last one in this series.

As always, you can download the current state of the application here : crossplatformqt-4.zip

Or, follow its Github repository here : pythonthusiast/CrossPlatformQt.

Stay tuned for my subsequent article!

PS : And I do apologize for bringing this article so late (it' been two week since my last article!). I think I've been cursed by the God of Perfection when saying similar things with, "In this article I will talk about how to develop Qt application using either C++, PyQt or PySide". Well, there you go.. A complete (I hope so) practical hands-on on those three technologies.

3 General Guide

General Guide on Qt GUI Application Development

As with any GUI application framework, developing GUI application using Qt GUI can be breakdown into these series of tasks:

Design the application user interface using provided GUI designer tool.

In this regard, you will use Qt Designer to design your Qt application user interfaces, saved in file having *.ui extension. The task of using Qt Designer can be made seamless, if we develop a C++ Qt application using Qt Creator. If however we develop Qt application using Python via PyQt/PySide, we must invoke Qt Designer manually. Another point to note that, using a GUI designer is actually an optional (but highly recommended) requirement. You can develop your application GUI bare handed using plain C++/Python code. But surely, using a visual GUI designer tool to laying out your application GUI elements is much more pleasant than manually hand coding them. If we are using GUI designer tool, there will be a process of importing our GUI designer file into our application code. Once again, if we are using a highly integrated IDE, this process will be seamless. You may not realize that this task ever existed, as with the case of developing C++ Qt application using Qt Creator. Developing it using Python though, will require us to invoke either PyQt pyuic4 tool or PySide pyside-uic to import our $*$.ui files into Python $*$.py code. If you have experience using Netbeans IDE to develop Java application, think of this process as the seamless process done by Netbeans when it automatically import Netbeans *.form file into Java code. Connecting certain events in objects GUI element/widget event into event handler. Well, using the term events in this particular case may not fit well in Qt world. But, I put a bet that most casual GUI programmer are more familiar with the term events than the term signal and slot used in Qt. For example, when you want to respond to a click of a button, what do you have in mind? Handling click events or connecting click signal with a slot function/method? Now, you got my point... Laughing Qt indeed have the term events, and together with signal and slot both did have the same purpose : object interconnection. However, the way event and signal/slot was carried out are an important matter that differentiate this two concepts: Event is implemented as a virtual method, where as slot is a function that can exist either as an object method or as a simple function. In this regard, Qt design event to be utilize by means of object inheritance, where as signal and slot is to be implemented as a loosely coupled connection between two object. There are chains of events that eventually can be cut off by our overridden method, where as after emitting signal, any functions that already connected as the receiving slot can respond in parallel, without regard of the other slot. Events is carried out from within system events loop, making it able to receive external system events such as mouse movement or keyboard strokes. This means an event handler may took sometime to execute after event did occur. This is significantly different with signal and slot, where slot function of a connected signal is –usually– carried out immediately after a signal is emitted. Once again, having use a good IDE can simplify your life of connecting the available signal from a widget to a user defined slot. Yes, correct, I am talking about QtCreator Smile Defining resource file and importing it to our code. Qt has its own resource file saved in an *.qrc file which is actually a regular XML file. This is a cross platform resource file where you can define files and text resource for your application. In C++ project, the task of importing *.qrc file into *.cpp file is carried out by Qt rcc tool. This tool is called automatically by Qt Creator, where as its PyQt counterpart, pyrcc4, or its PySide counterpart, pyside-rcc, must be called manually for a Python project.

4 Preparing Qt, PyQt and PySide in OSX using Homebrew

Using Homebrew to take the role as the OSX package manager is probably the effortless way to install Python/PyQt/PySide. If Homebrew works correctly on your OSX machine, then we can say that preparing a working Python environment in OSX is way a lot easier than its Windows counterpart. It may be on a par with Ubuntu with its apt package manager. If it is as simple as doing apt-get install in Ubuntu, then it is also as simple as doing brew install on OSX. How cool is that?

Anyway, although you can inspect that OSX already shipped with Python environment (Python 2.7.2 in my Mountain Lion), but the subsequent task of upgrading/installing Python packages is not officially supported by Apple. Therefore, using Homebrew (or simply Brew) to manage our OSX packages, is simply irresistible. And it's always better to left Apple built-in python distribution to be untouched.

Now lets move on to the task of preparing Python working environment in our OSX system using brew.

First, lets install brew using this command :

```
ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

Brew installation is very informative and a pleasant to work with. Just execute it and follow any error messages (if any), informational message or recommendation of how to make brew working as effective as possible in your OSX environment. Anyway, don't worry about that ruby thing. OSX is also shipped with ruby distribution. So, after waiting for couple minutes, soon you will have a working brew binary ready in your OSX environment.

Second, you can go ahead and install any Python and its packages using the command brew install *FORMULA*. Formula is just another name for package. To search for available package, simply issue the command brew search *FORMULA*. So, to find out whether we can install qt using brew, simply issue the command brew search qt. It will give you qt, qt5, pyqt and pyqt5 amongst other things.

Recall that PyQt5 is distributed only as Python 3 package, so if you want to use PytQt4, you will end up installing Python 2.7 also. This means, eventually you will have two different version of Python. Don't worry about it though. Brew will take care of this for you.

To install Qt4, PyQt4 and PySide, issue the command brew install python qt pyqt pyside pyside-tools.

To install Qt5 and PyQt5, issue the command brew install python3 qt5 pyqt5.

Installing Qt4 or Qt5 using brew will not install QtCreator, which is an important factor for a successful C++ Qt project development. To finish the preparation you can download QtCreator manually from here. In my OSX workstation I install the full binary distribution of Qt 5.2.1 along with its QtCreator. I also install PyQt4/PyQt5/PySide along with Qt4/Qt5 using brew, to allow PyQt/PySide available in my Python 2/3 environment.

There is one last requirement should you want to develop C++ Qt application in OSX : you must do a full installation of XCode. Yep, that dreaded 1.6GB of $*$.dmg file must be downloaded and installed in your machine. I have try to install the XCode command line tools only, but QtCreator still need the full installation of XCode in order for it to able to build and package our C++ Qt application. The good news is you can download XCode freely once you register as Apple Developer (which is also free). I purposely upgrade my Mountain Lion 10.8.2 into 10.8.4 to let me install the latest XCode 5.0.2. It works well with the latest QtCreator shipped with Qt 5.2.1.

5 Qt Creator

From http://www.qt.io/download/, download Qt5 download. Then

qt-opensource-mac-x64-1.6.0-7-online.dmg

will be downloaded. If you install it, Qt Creator will be installed.

```
jasminesongspro:~ jesong1126$ brew install python3 qt5 pyqt5
==> Downloading https://downloads.sf.net/project/machomebrew/Bottles/python3-3.4.2_1.maver
######################################################################
100.0%
==> Pouring python3-3.4.2_1.mavericks.bottle.1.tar.gz
==> Caveats
Pip has been installed. To update it
  pip3 install --upgrade pip

You can install Python packages with
  pip3 install <package>

They will install into the site-package directory
  /usr/local/lib/python3.4/site-packages

See: https://github.com/Homebrew/homebrew/blob/master/share/doc/homebrew/Homebrew
```

```
-and-Python.md

.app bundles were installed.
Run ‘brew linkapps‘ to symlink these to /Applications.
Error: The ‘brew link‘ step did not complete successfully
The formula built, but is not symlinked into /usr/local
Could not symlink bin/2to3-3.4
Target /usr/local/bin/2to3-3.4
already exists. You may want to remove it:
  rm /usr/local/bin/2to3-3.4

To force the link and overwrite all conflicting files:
  brew link --overwrite python3

To list all files that would be deleted:
  brew link --overwrite --dry-run python3

Possible conflicting files are:
/usr/local/bin/2to3-3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/2to3-3.4
/usr/local/bin/easy_install-3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/easy_
install-3.4
/usr/local/bin/idle3 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/idle3
/usr/local/bin/idle3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/idle3.4
/usr/local/bin/pip3 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/pip3
/usr/local/bin/pip3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/pip3.4
/usr/local/bin/pydoc3 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/pydoc3
/usr/local/bin/pydoc3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/pydoc3.4
/usr/local/bin/python3 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/python3
/usr/local/bin/python3-config -> /Library/Frameworks/Python.framework/Versions/3.4/bin/
python3-config
/usr/local/bin/python3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/python3.4
/usr/local/bin/python3.4-config -> /Library/Frameworks/Python.framework/Versions/3.4/bin/
python3.4-config
/usr/local/bin/python3.4m -> /Library/Frameworks/Python.framework/Versions/3.4/bin/python3.4m
/usr/local/bin/python3.4m-config -> /Library/Frameworks/Python.framework/Versions/3.4/bin/
python3.4m-config
/usr/local/bin/pyvenv -> /Library/Frameworks/Python.framework/Versions/3.4/bin/pyvenv
/usr/local/bin/pyvenv-3.4 -> /Library/Frameworks/Python.framework/Versions/3.4/bin/pyvenv-3.4
==> /usr/local/Cellar/python3/3.4.2_1/bin/python3 -m ensurepip --upgrade
==> Summary
??  /usr/local/Cellar/python3/3.4.2_1: 3866 files, 67M
==> Downloading https://downloads.sf.net/project/machomebrew/Bottles/qt5-
5.3.2.mavericks.b
######################################################################
100.0%
==> Pouring qt5-5.3.2.mavericks.bottle.1.tar.gz
==> Caveats
We agreed to the Qt opensource license for you.
If this is unacceptable you should uninstall.

This formula is keg-only, which means it was not symlinked into /usr/local.

Qt 5 conflicts Qt 4 (which is currently much more widely used).
```

```
Generally there are no consequences of this for you. If you build your
own software and it requires this formula, you'll need to add to your
build variables:

    LDFLAGS:  -L/usr/local/opt/qt5/lib
    CPPFLAGS: -I/usr/local/opt/qt5/include


.app bundles were installed.
Run 'brew linkapps' to symlink these to /Applications.
==> Summary
??  /usr/local/Cellar/qt5/5.3.2: 5737 files, 179M
==> Installing pyqt5 dependency: sip
==> Downloading https://downloads.sf.net/project/pyqt/sip/sip-4.16.3/sip-
4.16.3.tar.gz
######################################################################
100.0%
==> python configure.py --deployment-target=10.9 --destdir=/usr/local/Cellar/sip/4.16.3/li
==> make
==> make install
==> make clean
==> python3 configure.py --deployment-target=10.9 --destdir=/usr/local/Cellar/sip/4.16.3/l
==> make
==> make install
==> make clean
==> Caveats
The sip-dir for Python is /usr/local/share/sip.

Python modules have been installed and Homebrew's site-packages is not
in your Python sys.path, so you will not be able to import the modules
this formula installed. If you plan to develop with these modules,
please run:
  mkdir -p /Users/jesong1126/.local/lib/python2.7/site-packages
  echo 'import site; site.addsitedir("/usr/local/lib/python2.7/site-
packages")' >> /Users/jesong1126/.local/lib/python2.7/site-
packages/homebrew.pth
==> Summary
??  /usr/local/Cellar/sip/4.16.3: 12 files, 832K, built in 8 seconds
==> Installing pyqt5
==> Downloading https://downloads.sf.net/project/pyqt/PyQt5/PyQt-5.3.2/PyQt-gpl-5.3.2.tar.
######################################################################
100.0%
==> python3 configure.py --confirm-license --bindir=/usr/local/Cellar/pyqt5/5.3.2/bin --de
==> make
==> make install
==> make clean
??  /usr/local/Cellar/pyqt5/5.3.2: 693 files, 21M, built in 3.4 minutes
jasminesongspro:~ jesong1126$
jasminesongspro:~ jesong1126$ brew link --overwrite python3
```

6 Developing Qt Widgets Application with PyQt

General Guide on Developing Qt Application Using Python

Foreword warning : so far, there is no Python IDE equivalent in Qt development features compare to QtCreator. You will have to go back and forth between your Python IDE and QtDesigner to design application GUI interfaces, manually issue all counterpart command line utility of uic (import QtDesigner *.ui file into Python *.py code) and rcc (import Qt resource file *.qrc into Python *.py file) in its PyQt/PySide version and connect manually all your signal widgets to slot function. It lead me to think that, if there is no strict Python requirement in your Qt project, just use C++. Live can be so much easier. Or, in another perspective, imagine if someday they make Python support directly accessible in QtCreator. Now that's a simple life...

Anyway, in this PyQt project (and the later PySide version) we are going to use PyCharm Community Edition for Mac OS X as our Python IDE. It's a great IDE, despite its lack of tight support in PyQt/PySide application development. But then again, there is no Python IDE having that kind of support... So, PyCharm CE suit us just well...

7 Gui and Py

```
$ pyuic5 gs3.ui -o gs3.py
$ python3 gs3.py
```

7.1 sip

SIP Reference Guide

7.1.1 Downloading

You can get the latest release of the SIP source code from
http://www.riverbankcomputing.com/software/sip/download.

7.1.2 Configuring

Unpack the source package (either a .tar.gz or a .zip file depending on your platform) and goto the unpacked sip folder.

```
$ python configure.py
```

7.1.3 Building

```
$ make
$ sudo make install
```

7.2 PyQt5

PyQt5 Reference Guide

7.2.1 Downloading PyQt5

You can get the latest release of the PyQt5 source code from
http://www.riverbankcomputing.com/software/pyqt/download5.

7.2.2 Configuring PyQt5

Unpack the source package (either a .tar.gz or a .zip file depending on your platform) and goto the unpacked PyQt5 folder.

```
$ python configure.py --qmake  /Users/jesong1126/Qt/5.3/clang_64/bin/qmake
```

7.2.3 Building

```
$ make -j 4
$ sudo make install
```

7.3 QtDesigner

You can get the latest release of the QtDesigner source code from http://qt-project.org/downloads.

```
$ pip3 install pyqt_gs3
$ pyuic5 gs3.ui -o gs3.py
$ python3 gs3.py
$ chmod +x gs3.py
$ pip3 install bibtexvcs
$ python3 -m bibtexvcs.gui

$ pip3 install pandas
$ python3 -m pip install pandas
```