PySide GUI Application Development

Develop more dynamic and robust GUI applications using an open source cross-platform UI framework

Venkateshwaran Loganathan

PySide GUI Application Development

Develop more dynamic and robust GUI applications using an open source cross-platform UI framework

Venkateshwaran Loganathan

[PACKT]
PUBLISHING

PySide GUI Application Development

Credits

About the Author

Venkateshwaran Loganathan is an eminent software developer who has been involved in the design, development, and testing of software products for more than five years. He was introduced to computer programming at the age of 11 with FoxPro and then started to learn and master various computer languages like C, C++, Perl, Python, node.js, and Unix shell scripting. Fascinated by open source development, he has contributed to various open source technologies. He is now working for Cognizant Technology Solutions as an Associate in Technology, where he has involved himself in research and development for Internet of Things domain. He is now actively involved in using RFID devices for evolving the Future of Technology concepts. Before joining Cognizant, he worked at Infosys, Virtusa, and NuVeda. Starting his career as a Network Developer, he has gained expertise in various domains like Networking, e-learning, and HealthCare. He has won various awards and accolades for his work.

He holds a bachelor's degree in Computer Science and Engineering from Anna University and currently pursuing M.S in software systems from BITS, Pilani. Apart from programming, he is actively involved in handling various technical and soft skills classes for the budding engineers and college students. His hobbies are singing and trekking. He likes to get involved with social media. Visit him online at `http://www.venkateshwaranloganathan.com` and write to him at `anandvenkat4@gmail.com`.

About the Reviewers

Oscar Campos is a senior Python developer in Dublin, Ireland. He has great experience working with Python and many free software projects and contributions as well. He is the author of *SublimePySide the Sublime Text plugin to work with PySide*. You can take a look at his free software projects on his Github site on `https://github.com/DamnWidget`. He works for for Dedsert Ltd. an online gambling startup company located in south Dublin.

> I want to thank my wife Lydia for encouraging me to do this technical review

Dr. Jibo He is an avid developer using Python, PySide, and Qt. He has over seven years' experience using Python for his scientific research and entrepreneur careers. He has used PySide to develop important usability and search engine optimization software for the company UESEO LLC. He is honored to be the reviewer of this book, and expects more developers using PySide. He has also worked on PsychoPy/Python for Social Scientists and MATLAB for Social Scientists.

www.PacktPub.com

Support files, e-books, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers e-book versions of every book published, with PDF and ePub files available? You can upgrade to the e-book version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the e-book copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and e-books.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

The aim of this book is to introduce you to developing GUI applications in an easy way. Python is easy to learn and use and its programs are relatively shorter than those written in any other programming languages like C++, Java. It is supported by a large set of dynamic libraries and bindings that make it efficient to develop very complex applications in an efficient manner. This book will introduce you to user interface programming and its components. You will be able to develop real time applications in a shorter time after reading this book.

What this book covers

Chapter 1, *Getting Started with PySide*, introduces you to GUI programming in general. This chapter takes you through the introduction of PySide and its installation in various major operating systems followed by a short introduction to exception handling in programming. By the end of this chapter, users will know how to install and use PySide to create GUI applications in Python.

Chapter 2, *Entering through Windows*, introduces you with all the GUI programming that revolves around Windows. This chapter explains the basic methods of creating Windows and adding some functions to it. By the end of this chapter, users familiar in creating Windows and modify it accordingly.

Chapter 3, *Main Windows and Layout Management*, extends the previous chapter by explaining how to create menus and tool bars for a windowed application. This also explains the layout management policies. A simple text editor is given as an example at the end of the chapter. By the end of this chapter, readers have an experience of creating a real time application in PySide.

Chapter 4, *Events and Signals*, this chapter goes on explaining the signals, various text and graphic effects, drag-and-drop, and few geometrical diagram shapes. By the end of this chapter, readers will learn about managing events and various other text and graphical effects.

Chapter 5, *Dialogs and Widgets*, details the built-in dialog boxes for applications, introduces how to create customized dialogs, and then take a look at the various widgets available in PySide. By the end of this chapter, you will learn about creating your own customized widgets and dialogs.

Chapter 6, *Handling Databases*, explains how connecting to a database is evident for almost all applications. This chapter is dedicated to explaining how to connect to a database and execute queries on it. It also deals with the presentation of data in table and form views. By the end of this chapter, you will know more about interacting with databases and viewing data from them.

What you need for this book

To execute the examples provided in this book, you will require a standard installation of Python v2.6 or later and PySide v1.0.7 or later. A good text editor application like Sublime Text will also help in writing Python programs in an IDE environment.

Who this book is for

Are you a GUI developer or fascinated by GUI programming? Bored with writing several lines of code for creating a simple button in GUI? Then, this book is for you. The book is written for Python programmers to try their hands at GUI programming. Even if you are new to Python, but have some programming experience with any of the object oriented languages, you will be able to easily pick up since Python is easy to learn.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
try : suite
except(exception-1, exception-2, ... , exception-n) as target:
  suite
except : suite
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
...
myWindow.setIcon()
myWindow.setIconModes()
myWindow.show()
...
```

Any command-line input or output is written as follows:

```
sudo apt-get install python-pyside
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started with PySide

About Python

Python is a general-purpose, interpreted, object-oriented, high-level programming language with dynamic semantics. It is one of the most preferred programming languages by the programmers due to its interpreted nature and elegant syntax.

The success of Python lies in its simple and easy to learn syntax and the support of a wide variety of modules and packages which encourage program modularity and code reuse. Being an interpreted language, there is no compilation step which makes the edit-test-debug cycle incredibly fast paving the way to **Rapid Application Development**, the need of the hour. The support of the object-oriented features and high-level data structures, such as generators and list comprehensions, make Python a superior language to code small scripting programs to more advanced game programming.
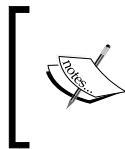
> Python is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with reptiles. Thus, making references to Monty Python skits in documentation is practiced and encouraged.

This book assumes that you have been acquainted with Python and want to test its capability in creating the GUI applications. However, Python is easy enough to learn in a week's time. If you know programming, then learning Python is a cakewalk for you. There are many resources available online and offline covering a wide range of topics. Being an open source language, Python is also supported by many programmers around the globe in the IRC system under the tag `#python`.

> **IRC** (**Internet Relay Chat**) provides a way of communicating in real time with people all over the world. Many open source programmers will be available online in the IRC chat under various channels to help you develop and contribute to the open source systems.

What is PySide?

Many modern programming languages are backed up by a set of libraries (commonly referred as toolkits) for creating GUI applications such as **Qt**, **Tcl/Tk**, **wxWidgets**. PySide is a Python binding of the cross-platform GUI toolkit Qt, and runs on all platforms supported by Qt including Windows, Mac OS X, and Linux. It is one of the alternatives used for GUI programming in Python to Tkinter.

PySide combines the advantages of Qt and Python. A PySide programmer has all the power of Qt, but is able to exploit it with the simplicity of Python. PySide is licensed under the LGPL Version 2.1 license, allowing both free/open source software and proprietary software development. PySide is evolving continuously as any other open source product and you are free to contribute to its development. Some of the applications such as Matplotlib, PhotoGrabber, Wing IDE, Lucas Chess, Fminer, and certify the wide spread usage of PySide in the software industry.

> The IRC channel for PySide is `#pyside` at Freenode.

There is some good news for mobile developers. Qt Mobility is a project creating a new suite of Qt APIs for mobile device functionality. The project "PySide Mobility" is a set of bindings that allows Python to access Qt Mobility API. So, learning PySide will also help you in learning mobile development. Come on, let's get started!

Hello GUI

In computing terms, GUI (pronounced "gooey") is used to notify a set of interfaces with computing systems, that involve user-friendly images rather than boring text commands. GUI comes as a rescue to the numerous command line interfaces that have a steep learning curve which requires a lot of dedication. Moreover, GUI systems make it easy for the end users to fulfill their needs without knowing much about the underlying commands which are unnecessary for them.

Every other application in the modern world is designed with excellent and interactive graphics to attract the end users. **Simplicity** and **Usability** are the two main ingredients for a successful GUI system. The demanding feature of a GUI is to allow the user to concentrate on the task at hand. To achieve this, it must make the interaction between the human and the computer seamless. Therefore, learning to program in GUI will not only make you a successful developer but also help in getting some revenue for yourself.

At a very basic level, GUI is seen as a window consisting of the following parts: controls, menu, layout, and interaction. A GUI is represented as a window on the screen and contains a number of different controls. Controls can, for example, be labels, buttons, or textboxes. Under the top frame of the GUI window, a menu bar is highly likely to be present giving users some choices to control the application. The top frame can also have buttons for hiding, resizing, or destroying the windows which are again controls. The controls are positioned in a certain layout which is very important in a good GUI design. The interaction happens in the way of I/O devices such as mouse and keyboard. Developing GUI application revolves around defining and controlling these components and designing the area of interaction is the most challenging part of all. The correct exploitation of events, listeners, and handlers will help in developing better GUI applications. Many frameworks have been developed to support GUI development such as PySide to assist programmers that make the GUI programming easy and quick. A good user interface design relates to the user, not the system architecture.

Setting up PySide

This is your first step in this series of learning. PySide is compatible with Python 2.6, Qt 4.6 or their later versions. So, before getting to install PySide, we must make sure that the minimum version compatibility is achieved. This section will teach you two ways of installing PySide. One, being the most common and easy way, is using simple point-and-click installers and package managers. This will install a most stable version of PySide in your system which you can use comfortably without worrying too much about the stability. However, if you are an advanced programmer, you may prefer to build PySide from scratch, from the latest builds available when you are reading this book. Both these methods are explained here for Windows, Mac OS X, and Linux systems, and you are free to choose your own setup style.

Installation

This section explains the ways to install PySide on Windows, Linux, and Macintosh operating systems.

Windows

Installation of PySide on Windows is easy with the help of installer. Follow these steps for setting up PySide on Windows:

1. Get the latest stable package compatible to your operating system architecture and the Python version installed from the releases page:

   ```
   http://qt-project.org/wiki/PySide_Binaries_Windows
   ```

2. Run the downloaded installer executable which will automatically detect the Python installation from your system.

3. You are given an option to install PySide on the default path or at the path of your choice.

4. Click on **Next** in the subsequent windows, and finally on **Finish**, PySide is installed successfully on your system.

Mac OS X

The binaries for MAC OS X installers of PySide are available at:

```
http://qt-project.org/wiki/PySide_Binaries_MacOSX
```

Download the latest version compatible with your system and carry out installation as explained earlier.

You can also choose to install the PySide from the command line with the help of Homebrew or by using MacPorts. The commands are:

```
brew install pyside
port-install pyXX-pyside


replacing XX with your Python version
```

Linux

Installing PySide on a Debian-based system is much easier with the synaptic package manager. Issuing the following command will fetch and install the latest stable version available in the aptitude distro:

```
sudo apt-get install python-pyside
```

On an RPM based system, you can use the RPM-based distro yum:

```
yum install python-pyside pyside-tools
```

> If you want to make sure that PySide is installed properly in your system, issue the following commands in the Python shell environment as shown in the following figure:
>
> `import PySide`: This should not return any errors
>
> `PySide.__version__`: This should output something like 1.1.2

A sample output is given for your reference is shown as follows:

Building PySide

This section explains how to build PySide in Windows, Linux, and Macintosh operating systems.

Windows

Before starting to build PySide on Windows, ensure that the following prerequisites are installed:

- Visual Studio Express 2008 (Python 2.6, 2.7, or 3.2) / Visual Studio Express 2010 (Python 3.3). Find results from `http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products`.

- Qt 4.8 libraries for Windows from `http://releases.qt-project.org/qt4/source/qt-win-opensource-4.8.4-vs2008.exe`.

- CMake from `http://www.cmake.org/cmake/resources/software.html`.

- Git from `http://git-scm.com/download/win`.
- Python 2.6, 2.7, 3.2, or 3.3 from `http://www.python.org/download/`.
- OpenSSL from `http://slproweb.com/products/Win32OpenSSL.html` (Optional).

Make sure that Git and CMake executable is set in your system PATH. Now, follow these steps to start building PySide:

1. Git clone the PySide repository from Github:

   ```
   c:/> git clone https://github.com/PySide/pyside-setup.git pyside-setup
   ```

2. Change your working directory to "pyside-setup":

   ```
   c:/> cd pyside-setup
   ```

3. Build the installer:

   ```
   c:\> c:\Python27\python.exe setup.py bdist_wininst --msvc-version=9.0 --make=c:\Qt\4.8.4\bin\qmake.exe --openssl=c:\OpenSSL32bit\bin
   ```

4. Upon successful installation, the binaries can be found in the sub-folder "dist":

   ```
   c:\pyside-setup\dist
   ```

On completion of these steps, the PySide should have been successfully built on your system.

Linux

To build PySide on Linux, the following prerequisites must be available. Check if you have them already or download them using the following links.

Prerequisites

- CMake >= 2.6.0: `http://www.cmake.org/cmake/resources/software.html`
- Qt libraries and development headers >= 4.6: `http://origin.releases.qt-project.org/qt4/source/qt-everywhere-opensource-src-4.8.4.tar.gz`
- libxml2 and development headers >= 2.6.32: `http://www.xmlsoft.org/downloads.html`
- libxslt and development headers >= 1.1.19: `http://xmlsoft.org/XSLT/downloads.html`
- Python libraries and development headers >= 2.5: `http://www.python.org/download/`

Building PySide

PySide is a collection of four interdependent packages API Extractor, Generator Runner, Shiboken Generator, and Pyside Qt bindings. In order to build PySide, you have to download and install the packages mentioned previously in that order.

- **API Extractor**: A set of libraries used by the bindings generator to parse the header and typesystem files to create an internal representation of the API. [`https://distfiles.macports.org/apiextractor/`]

- **Generator Runner**: Program that controls the bindings generation process according to the rules given by the user through headers, typesystem files, and generator frontends. It is dependent on the API Extractor. [`https://distfiles.macports.org/generatorrunner/`]

- **Shiboken Generator**: Plugin that creates the PySide bindings source files from Qt headers and auxiliary files (typesystems, global.h, and glue files). It is dependent on Generator Runner and API Extractor. [`https://distfiles.macports.org/py-shiboken/`]

- **PySide Qt Bindings**: Set of typesystem definitions and glue code that allows generations or a generation of Python Qt binding modules using the PySide tool chain. It is dependent on Shiboken and Generator Runner. [`http://download.qt-project.org/official_releases/pyside/`]

Always make sure that you are downloading the packages mentioned previously and building them in the same order as mentioned, since each of the packages are interdependent. Follow these steps to build the packages:

1. Unzip the downloaded packages and change into the package directory:

   ```
   tar –xvf <package_name>
   cd <package_directory>
   ```

2. Create a build directory under the package directory and enter that directory:

   ```
   mkdir build && cd build
   ```

3. Make the build using CMake:

   ```
   cmake .. && make
   ```

4. On successful make, build and install the package:

   ```
   sudo make install
   ```

5. Note that you require sudo permissions to install the packages.

6. To update the runtime linker cache, issue the following command:

   ```
   sudo ldconfig
   ```

Once you follow these steps in order for each of the packages, the PySide should have been successfully built on your system.

Mac OS X

Building PySide on a Mac system is the same as building it on Linux, except that Mac needs XCode-Developer tools to be installed as a prerequisite. The other prerequisites and building procedures are same as Linux.

> If you are installing the libs in a non-default system directory (other than `/usr/local`), you might have to update the `DYLD_LIBRARY_PATH` by typing the following command:
> **`export DYLD_LIBRARY_PATH=~/my_dir/install/lib`**

Importing PySide objects

Congratulations on setting up PySide successfully on your system. Now, it's time to do some real work using PySide. We have set up the PySide and now want to use it in our application. To do this, you have to import the PySide modules in your program to access the PySide data and functions. Here, let's learn some basics of importing modules in your Python program.

There are basically two ways widely followed when importing modules in Python. First, is to use a direct `import <module>` statement. This statement will import the module and create a reference to the module in the current namespace. If you have to refer to things (functions and data) that are defined in module, you can use `module.function`. Second, is to use `from module import *`. This statement will import the module and create references in the current namespace to all public objects defined by that module. In this case, if you have to refer to things that are defined in module, you can simply use function.

Therefore, in order to use the PySide functions and data in your program you have to import it by saying either `import PySide` or `from PySide import *`. In the former case, if you have to refer to some function from PySide you have to prefix it with PySide like `PySide.<function_name>`. In the latter, you can simply call the function by `<function_name>`. Also, note that in the latter statement `*` can be replaced by specific functions or objects. The use of `*` denotes that we are trying to import all the available functions from that module. Throughout this book, I prefer to use the latter format except that instead of importing all the modules by *, the specific modules are imported by their names. This is done to avoid allocating memory for the modules that we don't use.

My first PySide application

Let's get our hands dirty with some real coding now. We are going to learn how to create our first and traditional "Hello World" application. Have a look at the code first and we will dissect the program line-by-line for a complete explanation of what it does. The code might look little strange to you at first but you will gain understanding as we move through.

```python
# Import the necessary modules required
import sys
from PySide.QtCore import Qt
from PySide.QtGui import QApplication, QLabel
# Main Function
if __name__ == '__main__':

    # Create the main application
    myApp = QApplication(sys.argv)

    # Create a Label and set its properties
    appLabel = QLabel()
    appLabel.setText("Hello, World!!!\n Look at my first app using
      PySide")
    appLabel.setAlignment(Qt.AlignCenter)
    appLabel.setWindowTitle("My First Application")
    appLabel.setGeometry(300, 300, 250, 175)

    # Show the Label
    appLabel.show()

    # Execute the Application and Exit
    myApp.exec_()
    sys.exit()
```

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

On execution, you get an output window as shown in the following screenshot:
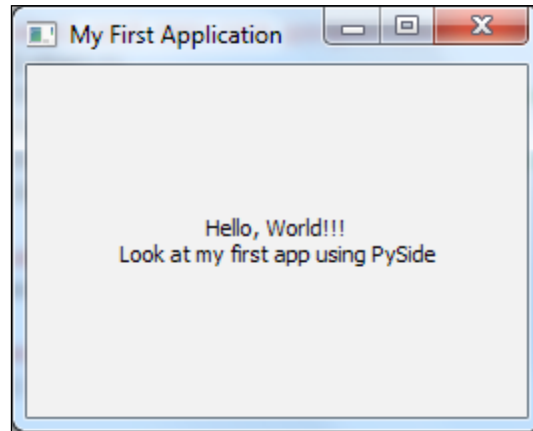
Now, let's get into the working of the code. We start with importing the necessary objects into the program.

The first three in the code snippet import the modules required for the program. Python is supported with a library of standard modules which are built into the interpreter and provide access to operations that are not part of the core language. One such standard module is `sys` which provides access to some variables and functions that are used closely by the interpreter. In the preceding program, we need sys module to pass command line arguments `sys.argv` as a parameter to the `QApplication` class. It contains the list of command line arguments passed to a Python script. In most of the GUI applications that use PySide, we might have two classes imported for a basic functionality. They are `QtCore` and `QtGui`. The `QtCore` module contains functions that handle signals and slots and overall control of the application whereas `QtGui` contains methods for creating and modifying various GUI window components and widgets.

In the main program, we are creating an instance of the `QApplication` class. `QApplication` creates the main event loop, where all events from the window system and other sources are processed and dispatched. This class is responsible for an application's initialization, finalization, and session management. It also handles the events and sets the application's look and feel. It parses the command line arguments (`sys.argv`) and sets the internal state accordingly. There should only be one `QApplication` object in the whole application even though it creates one or many windows at any point in time.

> The QApplication object must be created before any other objects as this handles the system-wide and application-wide settings for your application. It is also advised to create it before any modification of command line arguments received.

Once the main application instance is created, we move on by creating a QLabel instance that will display the required message on the screen. This class is used to display a text or an image. The appearance of the text or image can be controlled in many ways by the functions provided by this class. The two lines that follow the instantiation of this class set the text to be displayed and align it in a way that is centered on the application window.

Since Python is an object-oriented programming language, we take the advantage of many object-oriented features such as polymorphism, inheritance, object initialization, and so on. The complete Qt modules are designed in object-oriented paradigm that supports these features. QLabel is a base class inherited from the super class QFrame whose parent class is QWidget (the details will be covered in forthcoming chapters). So, the functions that are available in QWidget and QFrame are inherited to QLabel. The two functions setWindowTitle and setGeometry are functions of QWidget which are inherited by the QLabel class. These are used to set the title of the window and position it on the screen.

Now that all the instantiation and setup is done, we are calling the show function of the QLabel object to present the label on the screen. It is only at this point that the label becomes visible to the user on the screen. Finally, we call the exec_() function of the QApplication object which will enter the Qt main loop and start executing the Qt code. In reality, this is where the label will be shown to the user but the details can be safely ignored as of now. Finally, we exit the program by calling sys.exit().

Exception handling as a practice

It is always not possible to foresee all the errors in your programs and deal with them. Python comes with an excellent feature called **Exception Handling** to deal with all the runtime errors. The aim of the book is not to explain about this feature in detail but to give you some basic ideas so that you can implement it in the code that you write.

In general, the exceptions that are captured while executing a program are handled by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as exception handler. Once they are handled successfully, the program takes the normal execution flow by using the saved information. Sometimes, the normal flow may be hindered due to some exceptions that cannot be resolved transparently. In any case, exception handling provides a mechanism for the smooth flow of the program altogether.

In Python, the exception handling is carried out in a set of try and except statements. The try statements consists of a set of suspicious code that we might think cause an exception. On hitting an exception, the statement control is transferred to except block where we can have a set of statements that handles the exception and resolves it for a normal execution of a program. The syntax for the same is as follows:

```
try : suite
except(exception-1, exception-2, ... , exception-n) as target:
  suite
except : suite
```

where `suite` is an indented block of statements. We can also have a set of try, except block in a try suite. The former except statement provides a specific exception class which can be matched with exception that is raised. The latter except statement is a general clause which is used to handle the "catch-all" version. It is always advisable to write our code in the exception encapsulation.

In the previous example, consider that we have missed to instantiate the `appLabel` object. This might cause an exception confronting to a class of exception called "Name Error". If we don't encapsulate our code within the try block, this raises a runtime error. However, if we put our code in a try block, an exception can be raised and handled separately; this will not cause any hindrance to the normal execution of the program. The following code explains this with the possible output:

```
# Import the necessary modules required
import sys
from PySide.QtCore import *
from PySide.QtGui import *

# Main Function
if __name__ == '__main__':

    # Create the main application
    myApp = QApplication(sys.argv)
```
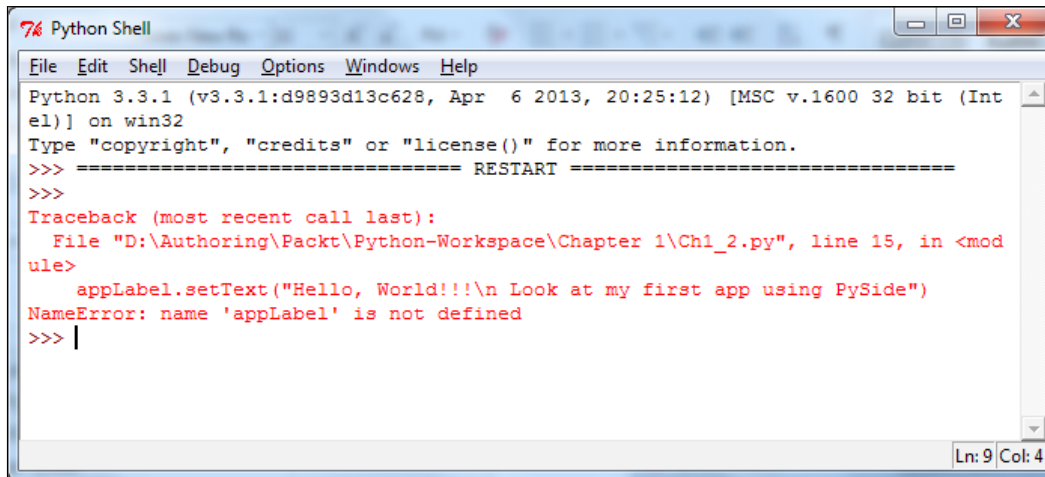
```
# Create a Label and set its properties
try:
    #appLabel = QLabel()
    appLabel.setText("Hello, World!!!\n Look at my first app
        using PySide")
    appLabel.setAlignment(Qt.AlignCenter)
    appLabel.setWindowTitle("My First Application")
    appLabel.setGeometry(300, 300, 250, 175)

    # Show the Label
    appLabel.show()

    # Execute the Application and Exit
    myApp.exec_()
    sys.exit()
except NameError:
    print("Name Error:", sys.exc_info()[1])
    pass
```

In the preceding program, if we don't handle the exceptions, the output would be as shown in the following screenshot:
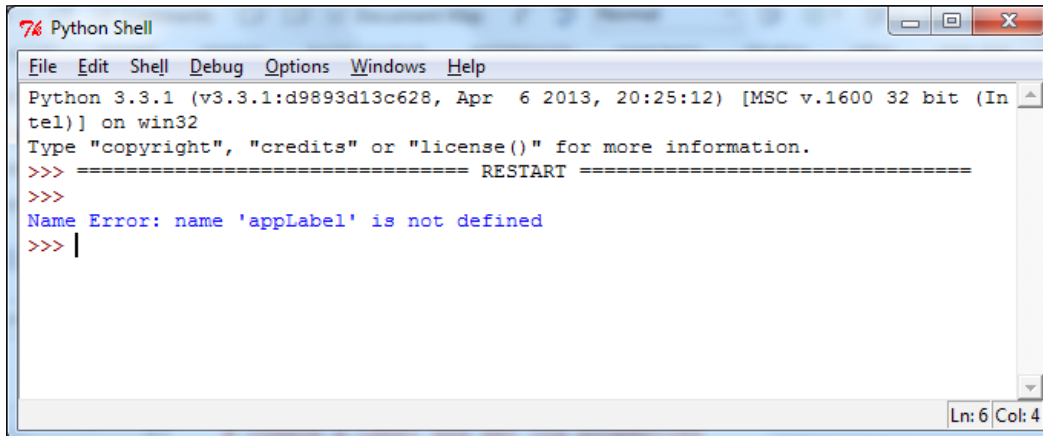
However, if we execute the preceding code, we would not run into any errors as shown in the following screenshot. Instead, we could have captured the exception and given some info about it to the user shown as follows:

Hence, it is always advised to implement exception handling as a good practice in your code.

Summary

The combination of Qt with Python provides flexibility to the Qt developers for developing GUI programs in a more robust language and also providing a rapid application development platform available on all major operating systems. We introduced you to the basics of PySide and its installation procedure on Windows, Linux, and Mac systems. We went on creating our first application which introduced the main components of creating a GUI application and the event loop. We have concluded this chapter with an introduction to exception handling as a best practice. Moving on, we are set to create some real-time applications in PySide.

2
Entering through Windows

The main part of any GUI program is to create windows and define functionalities around it. We will start exploring ways to create windows and customize it in this chapter and will move on to create a real-time windows application in the next chapter.

The widget is the center of the user interface. It receives the user inputs from mouse, keyboard, and other events, of the window system, and paints a representation of itself on the screen. Every widget is rectangular, and sorted in a **Z-order**. A widget is clipped by its parent and by the widgets in front of it. A widget that does not have a parent is called a window and is always independent. Usually, windows have a frame and a title bar at the least but it is possible to create them without these by setting some windows flags. This chapter explains how to create simple windows using QWidget and also how to create some widely used widgets. The code snippets that are explained from this chapter onwards will be based on **Object-Oriented Design principles**.

Creating a simple window

The `QWidget` is the base class for all the user interface classes. A widget can be a top-level widget or a child widget contained in a top-level or parent widget. Now, let's create a top-level window using `QWidget`. The constructor of the `QWidget` class takes two optional parameters, `parent` and `flags`. The parent can be a `QWidget` object and flags can be a combination of `PySide.QtCore.Qt.WindowFlags`.

```
# Import required modules
import sys
import time
from PySide.QtGui import QApplication, QWidget
```

```
class SampleWindow(QWidget):
    """ Our main window class
    """

    # Constructor function
    def __init__(self):
        QWidget.__init__(self)
        self.setWindowTitle("Sample Window")
        self.setGeometry(300, 300, 200, 150)
        self.setMinimumHeight(100)
        self.setMinimumWidth(250)
        self.setMaximumHeight(200)
        self.setMaximumWidth(800)


if __name__ == '__main__':
    # Exception Handling
    try:
        myApp = QApplication(sys.argv)
        myWindow = SampleWindow()
        myWindow.show()
        time.sleep(3)
        myWindow.resize(300, 300)
       myWindow.setWindowTitle("Sample Window Resized")
        myWindow.repaint()
        myApp.exec_()
        sys.exit(0)
    except NameError:
        print("Name Error:", sys.exc_info()[1])
    except SystemExit:
        print("Closing Window...")
    except Exception:
        print (sys.exc_info()[1])
```

In this sample program, we create a window, set its minimum and maximum size and repaint the window with different dimensions after a short period. If you look at the code closely, you may analyze that the code follows exception handling mechanism and object-oriented principles as explained earlier.

The main idea in the earlier program is to introduce you with creating classes and objects and work around them since programming in PySide indirectly implies programming using OO principles. PySide libraries follow the OO principles and so do we. Our sample window is instantiated with the class that we have declared for this purpose. The class `SampleWindow` is inherited from the `PySide.QtGui.QWidget` class. So, all the properties of `QWidget` can be applied to our `SampleWindow` class also. The `__init__` function is called the constructor to be shown when an object is instantiated. While instantiating the object, we also call the constructor function of its super class for proper instantiation and hence the line `QWidget.__init__(self)`. The methods that follow this line are all inherited from the `QWidget` class. The functions `setMinimumHeight`, `setMinimumWidth` set the window to minimum size and cannot be shrinked further. Similarly, the window cannot be extended beyond the maximum size specified by the functions `setMaximumHeight` and `setMaximumWidth`.

Our `main` function is encapsulated in a `try`, `catch` block to deal with any unexpected exceptions that may occur. As explained in the previous chapter, every PySide application must create a main application object. So, we start with creating an object for the `QApplication` class. Then, we create an object for our custom defined class `SampleWindow`. At this point, the `__init__` function is called and all the properties defined for our sample window are set. We paint the window on the screen by calling the show function on the `SampleWindow` object. The lines that follow are just an example to show that we can repaint the windows with different dimensions at any point during the execution of the program. So, we hold on (sleep) for 3 seconds, resize the window and repaint it on the screen. Now, execute the code and have some fun.

On executing the program, you will be shown a window as shown in the following image. This window will get resized after 3 seconds. Also, try to resize the window by dragging its corners. You may notice that the window cannot be shrunk or expanded beyond the minimum and maximum metrics set in our earlier code. You might not initially see a window when executing this program on an XWindow based system such as Linux, because the main application loop has not been called yet, so none of the objects has been really constructed and buffered out to the underlying XWindow system.
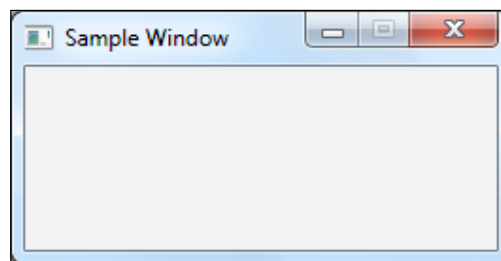
The following figure is the screenshot of the final output that you will see:
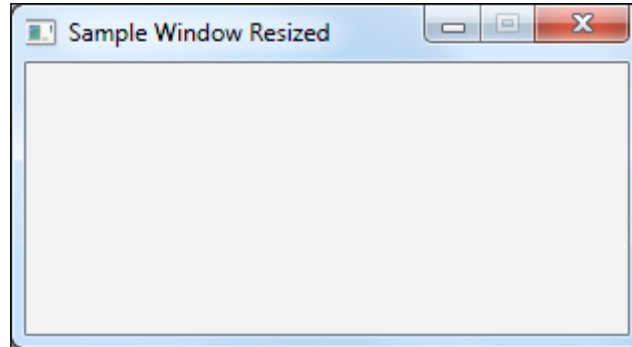
Creating the application icon

We have created our sample window and now we go on customizing it with some features for our needs. For each customization, we add a new function under the `SampleWindow` class in the previous program to define its properties and call that our main function to apply those properties on the sample window. In this section, we define an icon to be set on the window that we created. An icon is a small image that is created to visually emphasize the purpose of the program. It is displayed in the top-left corner of the application window. The same is also displayed in the taskbar when the application is minimized. As a prerequisite for this program, you might need an icon image with the dimensions similar to the image used here (72 X 72). You can create your own image or download it from the book site if you wish to use the one used in this program:

```
# Import required modules
import sys
from PySide.QtGui import QApplication, QWidget, QIcon

class SampleWindow(QWidget):
    """ Our main window class
    """
    def __init__(self):
        """ Constructor Function
        """
        QWidget.__init__(self)
        self.setWindowTitle("Icon Sample")
        self.setGeometry(300, 300, 200, 150)
```

```
      def setIcon(self):
          """ Function to set Icon
          """
          appIcon = QIcon('pyside_logo.png')
          self.setWindowIcon(appIcon)

if __name__ == '__main__':
    # Exception Handling
    try:
        myApp = QApplication(sys.argv)
        myWindow = SampleWindow()
        myWindow.setIcon()
        myWindow.show()
        myApp.exec_()
        sys.exit(0)
    except NameError:
        print("Name Error:", sys.exc_info()[1])
    except SystemExit:
        print("Closing Window...")
    except Exception:
        print(sys.exc_info()[1])
```

In the preceding program, we have included a class to set the application icon and we call that function from our main program to set it. Remember to place the image in the same location as the program. On executing this program, we would get the output as shown in the following screenshot:
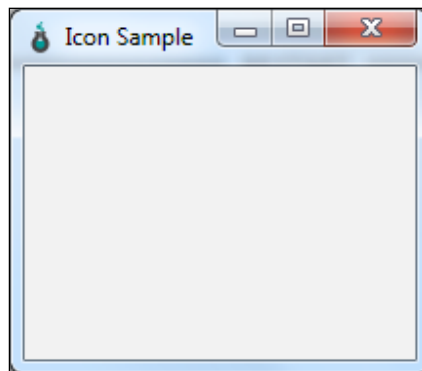
As we have seen the basics of setting an application icon, we move on to explore more about the `PySide.QtGui.QIcon` class. This class provides a set of functions that provides scalable icons in different modes and states. By using this class, we can create various types of icons differing in their size and mode, say; smaller, larger, active, and disabled from the set of `pixmaps` it is given. Such `pixmaps` are used by the Qt widgets to show an icon representing a particular action.

The `QIcon` class has the following different forms of constructors:

```
QIcon()
QIcon(engine)
QIcon(other)
QIcon(pixmap)
QIcon(filename)
```

The first form constructs a null icon. The second form takes `PySide.QtGui.QIconEngine` as a parameter. This class provides an abstract base class for the `QIcon` renderers. Each icon has a corresponding engine that is responsible for drawing the icon with the requested size, mode, and state. The third form simply copies from the other `QIcon` object and it is considered to be the fastest method of all. The fourth form constructs the icon from the `PySide.QtGui.QPixmap` class. This class is an off-screen image representation that can be used as a paint device. A `pixmap` can be easily displayed on the screen using `PySide.QtGui.QLabel` or one of the two button classes, `PySide.QtGui.QPushButton` or `PySide.QtGui.QToolButton`. `QLabel` has a pixmap property whereas `QPushButton`/`QToolButton` has an icon property. The last form constructs an icon from the given filename. If filename contains relative path, it must be relative to the runtime working directory.

Icons are not only used for showing as application icon but also in various places as tool representation in the toolbars. Consider, we are creating a toolbar in our application where we display icons to represent functionalities in pictorial form. A sample toolbar may appear like the one shown in the following screenshot:
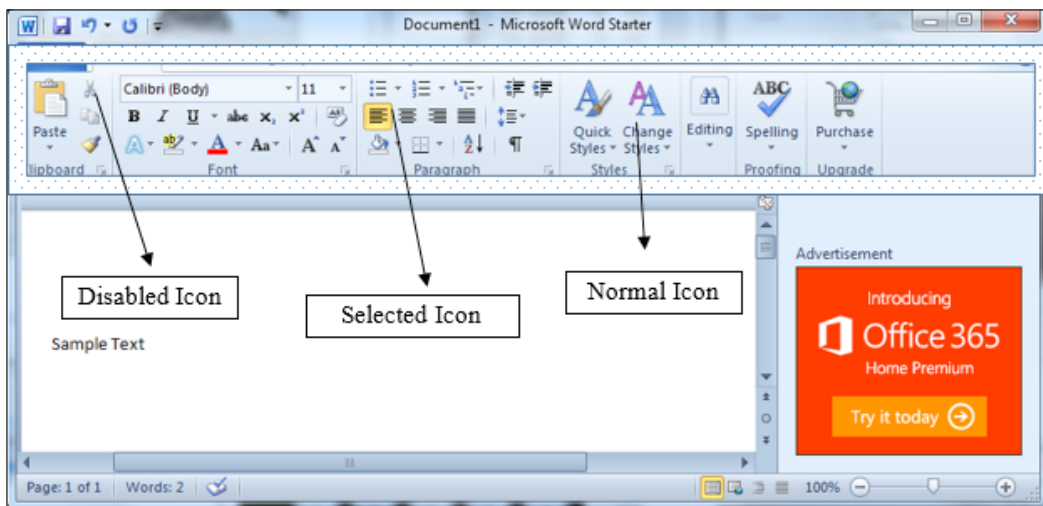
The `QIcon` class provides various modes to display the icon by the state it is defined by using the `pixmap` function applied to the `QIcon` class. The syntax of the `pixmap` function is `PySide.QtGui.QIcon.pixmap(width, height[, mode=Normal[, state=Off]])`. The parameters width and height represents the icon size. The modes can be any of the following four modes depending on the action:

Constant	Description
QIcon. Normal	Display the pixmap when the user is not interacting with the icon, but the functionality represented by the icon is available.
QIcon. Disabled	Display the pixmap when the functionality represented by the icon is not available.
QIcon.Active	Display the pixmap when the functionality represented by the icon is available and the user is interacting with the icon, for example, moving the mouse over it or clicking it.
QIcon. Selected	Display the pixmap when the item represented by the icon is selected.

The state parameter can be used to describe the state for which `pixmap` is intended to be used. It can take any of the following two values.

Constant	Description
QIcon.Off	Display the pixmap when the widget is in an "off" state
QIcon.On	Display the pixmap when the widget is in an "on" state

The following function will provide you with an example of various modes of icons that we create from setting the modes in the **pixmap** function. Add the following function from the previous program inside the `SampleWindow` class.

```
def setIconModes(self):
    myIcon1 = QIcon('pyside_logo.png')
    myLabel1 = QLabel('sample', self)
    pixmap1 = myIcon1.pixmap(50, 50, QIcon.Active, QIcon.On)
    myLabel1.setPixmap(pixmap1)

    myIcon2 = QIcon('pyside_logo.png')
    myLabel2 = QLabel('sample', self)
    pixmap2 = myIcon2.pixmap(50, 50, QIcon.Disabled, QIcon.Off)
    myLabel2.setPixmap(pixmap2)
    myLabel2.move(50, 0)

    myIcon3 = QIcon('pyside_logo.png')
    myLabel3 = QLabel('sample', self)
    pixmap3 = myIcon3.pixmap(50, 50, QIcon.Selected, QIcon.On)
    myLabel3.setPixmap(pixmap3)
    myLabel3.move(100, 0)
```

Now, add the following line in the main program to call this function:

```
...
myWindow.setIcon()
myWindow.setIconModes()
myWindow.show()
...
```

You might have noted a new widget `QLabel` used in this program. The `QLabel` widget is used to provide a text or image display. Running this program will output a window containing different modes of the same icon as shown in the following screenshot:
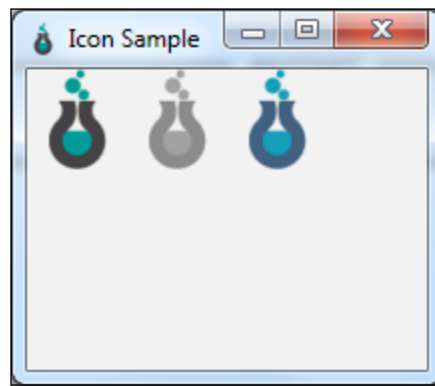
Showing a tooltip

Our next customization is to show a tooltip for the different modes of icons shown. Tooltip is handy when you need to show some help text or information to the users. Displaying help text for the widgets used in the window is an integral part for any GUI application. We use the `PySide.QtGui.QToolTip` class to provide tooltips (also called balloon help) for any widget. The `QToolTip` class define the properties of the tooltip like font, color, rich text display, and so on. As an example, the font properties can be set as:

```
QToolTip.setFont(QFont("Decorative", 8, QFont.Bold))
```

After setting the font, we set the tooltip to the widget by calling the `setToolTip()` function provided by the `QWidget` class:

```
myLabel1.setToolTip('Active Icon')
```

The `QFont` class specifies a font used for drawing the text. By using the functions provided by this class, we can specify various attributes that our font want to have. If the font type we specify is not installed or available, `Qt` will try to use the closest match available. If a chosen font does not include all the characters that need to be displayed, `QFont` will try to find the characters in the nearest equivalent fonts. When a `PySide.QtGui.QPainter` draws a character from a font the `PySide.QtGui.QFont` will report whether or not it has the character; if it does not, `PySide.QtGui.QPainter` will draw an unfilled square.

Modify the previous program as shown in the following code snippet:

```
def __init__(self):
        QWidget.__init__(self)
        self.setWindowTitle("Icon Sample")
        self.setGeometry(300, 300, 200, 150)
        QToolTip.setFont(QFont("Decorative", 8, QFont.Bold))
        self.setToolTip('Our Main Window')
    ...


def setIconModes(self):
...
        myLabel1.setPixmap(pixmap1)
        myLabel1.setToolTip('Active Icon')
        ...
        myLabel2.move(50, 0)
        myLabel2.setToolTip('Disabled Icon')
        ...
        myLabel3.move(100, 0)
        myLabel3.setToolTip('Selected Icon')
```

After making the changes, run the program and you can see the tooltip text is displayed as and when you move over the icon and window. The sample output is displayed in the following screenshot:
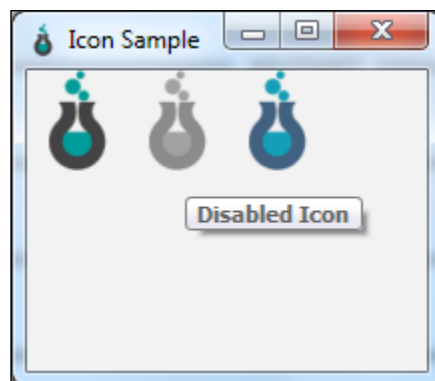
Adding a button

The next customization is to add a button to our application. The most common button used in any GUI is a push or command button. We push (click on) the button to command the computer to perform some action or answer a decision. Typical push buttons include **OK**, **Apply**, **Cancel**, **Close**, **Yes**, **No**, and **Help**. Usually the **push** button is rectangular with some label on it. It can also have an optional icon associated with it. It is also possible to define a shortcut key combination to the button to which it responds on clicking the key combination.

The button emits a signal when it is activated by any external event, say, mouse click or by pressing the spacebar or by a keyboard shortcut. A widget may be associated with this key click event which is executed on receiving this signal and it is usually called a slot, in *Qt*. We will learn more about signals and slots in the later chapters. As for now, be informed that a signal will connect to a slot on emit. There can also be other signals that are provided on a button like button pressed, button released, button checked, button down, button enabled, and so on. Apart from a push button we also have other button types in Qt like QToolButton, QRadioButton, QCommandLinkButton and QCheckBox will be discussed later.

QPushButton can be instantiated in three ways. It has three constructors with different signatures. They are:

```
QPushButton(parent=None)
QPushButton(text, [parent=None])
QPushButton(icon, text, [parent=None])
```

The parent parameter can be any widget, text is any string or a set of unicode characters, and icon is a valid QIcon object.

In this example program, we are going to add a button that will close the application when clicked. We define a button first and will call a function (*slot*) when clicked (*signal*).

```
def setButton(self):
    """ Function to add a quit button
    """
    myButton = QPushButton('Quit', self)
    myButton.move(50, 100)
    myButton.clicked.connect(myApp.quit)
```

Add the preceding function to the earlier example class and call the function from your __main__ conditional block before calling the `show()` function of myWindow. The important point here is the `clicked.connect()` call of the `myButton` object. The event `clicked` connects to the slot `myApp.quit()` which quits the application. The slot can be replaced by an excerpt of code or a user defined function which performs a set of operations.

It is highly likely that the quit button may be pressed by mistake. If the application is quit without the user's confirmation there is a high chance of it being a mistake. So, we are going to display a confirmation message to the user on clicking the quit button. If the user wishes to quit, the application quits or the user can cancel it. The widely used widget for this purpose is `QMessageBox`. This is used for providing a modal dialog box for informing the user or for asking the user a question and receiving an answer. We will see more in detail about the `QMessageBox` in *Chapter 5*, *Dialogs and Widgets*. Here, we just create an instance of it and add it to our program.

To do this, create a function as follows:

```
def quitApp(self):
    """ Function to confirm a message from the user
    """
    userInfo = QMessageBox.question(self, 'Confirmation',
       "This will quit the application. Do you want to Continue?",
     QMessageBox.Yes | QMessageBox.No)
    if userInfo == QMessageBox.Yes:
        myApp.quit()
    if userInfo == QMessageBox.No:
        pass
```

Now, change the connect function in the `setButton` module to call this function on click of the quit button:

```
myButton.clicked.connect(self.quitApp)
```

On executing the program and clicking on the quit button, you will get a confirmation message for which you can click on **Yes** or **No**:
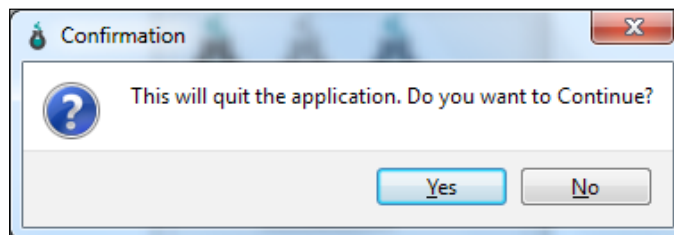
Clicking on **Yes** will quit the app and clicking on **No** will do nothing.

Centering the Window on the screen

Many windowed applications will get their importance by the way they are displayed on the screen. It is better practice for any GUI application to display the window centered on the screen. There are two advantages of doing this. One is to get the attention of the user and the other is to adjust with different display formats of various monitor screens.

There is no straightforward method to center the window by calling a predefined function of some class. Therefore, we write our own method called center to position the window center to any screen. The method takes the object it is calling and centers it with respect to the screen it is displayed:

```
def center(self):
    """ Function to center the application
    """
    qRect = self.frameGeometry()
    centerPoint = QDesktopWidget().availableGeometry().center()
    qRect.moveCenter(centerPoint)
    self.move(qRect.topLeft())
```

In order to do this, first we get the size and location of the window that we want to be centered. Then, we need to get the center point of the screen. Finally, we will move the window to the center of the screen. The `frameGeometry()` function will return a `PySide.QtCore.QRect` object which will hold the height, width, top, and left points of the window. The `QDesktopWidget().availableGeometry().center()` call will return the center point of the screen. The next two lines will move the window to the center point of the screen. Remember to call this function before the `myWindow.show()` line to view the settings applied on your window.

About box

Our next customization is to add an about box to our application. An about box is a dialog box that displays credits and revision information about the application. It may also include installed version and copyright information. The `QMessageBox` class provides a built-in function for this. It has the following signature:

```
PySide.QtGui.QMessageBox.about(parent, title, text)
```

The parent parameter takes any valid `QWidget` object. The title and text can be supplied with the Unicode parameter. The about box will have a single button labeled **OK**. To explain its working, we will now add a button named **About** and on click signal of that button we call a slot which will display the **About box** to the user. The code for the same is as follows:

```
def setAboutButton(self):
    """ Function to set About Button
    """
    self.aboutButton = QPushButton("About", self)
    self.aboutButton.move(100, 100)
    self.aboutButton.clicked.connect(self.showAbout)

 def showAbout(self):
    """ Function to show About Box
    """
    QMessageBox.about(self.aboutButton, "About PySide",
           "PySide is a cross-platform tool for generating GUI
Programs.")
```

Call this function from your __main__ block before the `myWindow.show()` call. On execution, you will see a window with an added **About** button. On click of this button, a separate dialog box is displayed with the heading and content text as given in the program. You can also note that the window also has the application icon displayed by default at the side. The sample output is displayed as follows for your reference:
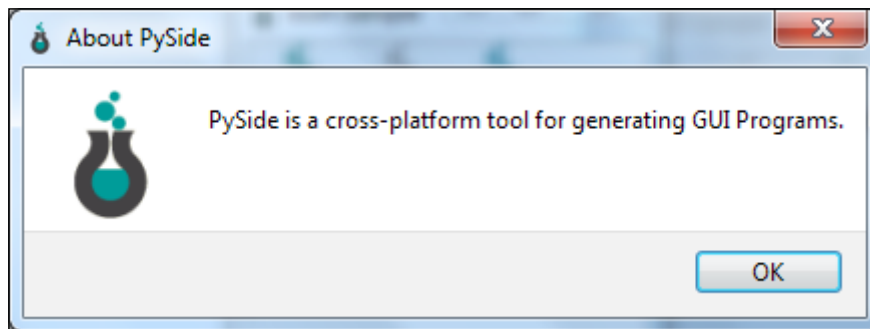
There is one more function provided by the `QMessage` class that might interest you. The `aboutQt` method is called whose signature is given as follows:

```
PySide.QtGui.QMessageBox.aboutQt(parent[, title=""])
```

Calling this function will display a simple message box about `Qt`, with the given title and centered over parent. The message will display the version of the `Qt` that is currently used by the application. It is useful for displaying information about the platform you are using in your program. As a learning exercise, try to implement this on your own and see what its output is.

Timers

Most of the GUI applications are time bound and it is extremely important to use timers to capture information about the program runtime and other similar tasks. You might also use timers to generate some event at specified time intervals, calculate the elapsed time for some action, to implement a countdown timer and so on. This section of the chapter covers how to create and use timers in our application and we develop a digital clock application explaining the concept of timers.

The classes that are used for creating this application in Qt are `PySide.QtCore.QTimer` and `PySide.QtCore.QDateTime`. The `QTimer` class provides high-level programming interface for timers. It provides repetitive and single-shot timers. Repetitive timers run continuously and restart at expiry of one time slot. Single-shot timers will run exactly once and expire after one time slot. A timeout event will occur at expiry of the given time slot. The timer can be started by issuing a start call to the `QTimer` object and can be stopped anywhere in between before the expiry of the time slot by issuing a stop signal:

```
QTimer.start(1000)
QTimer.stop()
```

The unit of timer is in milliseconds. The accuracy of timers depends on the underlying operating system and hardware. Most platforms support a resolution of 1 millisecond though the accuracy of the timer will not match this and is not guaranteed.

The `PySide.QtCore.QDateTime` class provides a calendar date and clock time functions. It is a combination of `PySide.QtCore.QDate` and `PySide.QtCore.QTime` classes. As with any other framework, the `QDateTime` class provides functions for comparing datetimes and for manipulation of a datetime. It provides a full set operator to compare two `QDateTime` objects where smaller means earlier and larger means later. The `QDateTime` can store datetimes both as local and as UTC. The `QDateTime.toUTC()` function can be applied is on a `QDateTime` object to convert the local time to UTC. This class handles and aware of daylight saving time:

```
# Import required modules
import sys
from PySide.QtCore import QDateTime, QTimer, SIGNAL
from PySide.QtGui import QApplication, QWidget, QLCDNumber
```

```python
class MyTimer(QWidget):
    """ Our Main Window class for Timer
    """
    def __init__(self):
        """ Constructor Function
        """
        QWidget.__init__(self)
        self.setWindowTitle('My Digital Clock')
        timer = QTimer(self)
        self.connect(timer, SIGNAL("timeout()"), self.updtTime)
        self.myTimeDisplay = QLCDNumber(self)
        self.myTimeDisplay.setSegmentStyle(QLCDNumber.Filled)
        self.myTimeDisplay.setDigitCount(8)
        self.myTimeDisplay.resize(500, 150)
        timer.start(1000)

    def updtTime(self):
        """ Function to update current time
        """
        currentTime = QDateTime.currentDateTime().toString('hh:mm:ss')
        self.myTimeDisplay.display(currentTime)

# Main Function
if __name__ == '__main__':
    # Exception Handling
    try:
        myApp = QApplication(sys.argv)
        myWindow = MyTimer()
        myWindow.show()
        myApp.exec_()
        sys.exit(0)
    except NameError:
        print("Name Error:", sys.exc_info()[1])
    except SystemExit:
        print("Closing Window...")
    except Exception:
        print(sys.exc_info()[1])
```

The preceding program will display a digital clock on execution. To display the time with the precision in seconds we start a timer that times out every second. On timeout of the timer, we call the `updtTime()` function which will update the current time and display it on the screen. In order to display the time in digital format we have used a special display in this program, which is different from the previous ones. The `PySide.QtGui.QLCDNumber` will display a number with LCD like digits which gives the appearance of a digital clock. The digits/numbers that can be shown with **LCDNumber** are **0/O, 1, 2, 3, 4, 5/S, 6, 7, 8, 9/g, minus, decimal point, A, B, C, D, E, F, h, H, L, o, P, r, u, U, Y, colon, degree sign** (which is specified as single quote in the string) and **space**. `PySide.QtGui.QLCDNumber` substitutes spaces for illegal characters. Using this, we can just output the text/number in any size. The `setSegmentStyle()` function sets the style of the `QLCDNumber` to be displayed, it could take the following values:
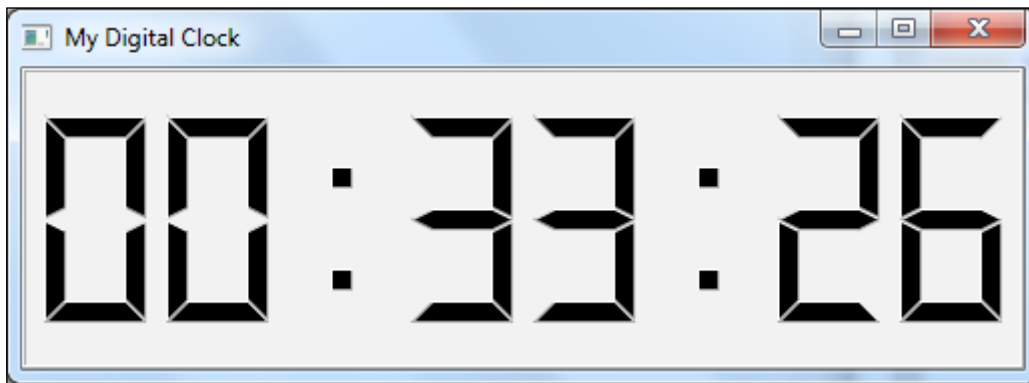
| Constant	Description
QLCDNumber.Outline	Gives raised segments filled with the background color.
QLCDNumber.Filled	Gives raised segments filled with the windowText color.
QLCDNumber.Flat	Gives flat segments filled with the windowText color.

One more thing to note here is the `setDigitCount` function which will set the number of digits to show on the display which defaults to five.

Windows style

The PySide application can be executed under different platforms/flavors of operating systems. The GUI style of each flavor may vary in representing the application. If you require your application to look good on all platforms, you have to style your application native to the operating system. Qt contains a set of style classes that emulate the styles of the different platforms. The abstract class that performs this in Qt is the `PySide.QtGui.QStyle` class. The classes that inherit this class and provide various style options are `QCommonStyle`, `QWindowsStyle`, `QPlastiqueStyle`, `QCleanlooksStyle`, `QGtkStyle`, `QMotifStyle`, and `QCDEStyle`. Qt's built-in widgets uses QStyle to perform nearly all of their drawing, ensuring that they look exactly like the equivalent native widgets. As an example, the following screenshot contains the different representation of combobox in eight different styles under various OS platforms:
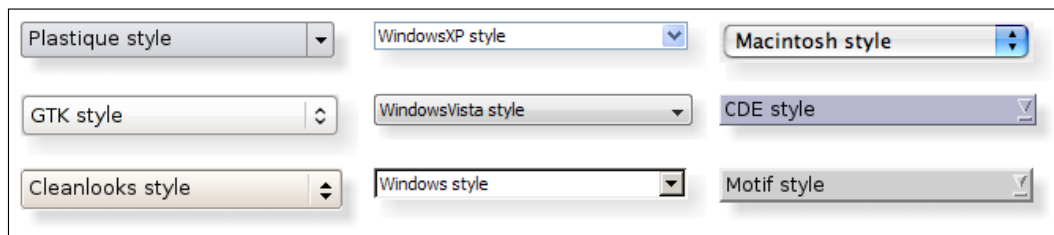
By default, Qt will choose the most appropriate style for the user's platform or desktop environment. The Windows style is the default style under flavors of Windows, Plastique style for Qt/X11 applications running under KDE, and cleanlooks is the default under GNOME. These styles use gradients and anti-aliasing to provide a modern look and feel. The style of the entire application can be set by using the `QApplication.setStyle()` function. It can also be specified by the user of the application, using the `-style` command line option while running the program. The command line option overrides the default style.

```
python myApp.py -style motif
```

The `setStyle()` function can be applied to an individual widget also. You can call `QApplication.setStyle()` any time, but by calling it before the constructor, you ensure that the user's preference, set using the `-style` command-line option, is respected.

Summary

In this chapter, we have seen how to create windows application using widgets and dialogs. We have also seen some customizations that are added to our application and finished the chapter with creating a digital clock application. We will start to explore creating main windowed applications in the coming chapters.

3
Main Windows and Layout Management

In the previous chapter, we have seen how to create windows using widgets. Most of the GUI applications that we use today are main window styled applications, which have a menu bar, toolbars, a status bar, a central widget, and optional dock widgets. Generally, an application will have a single main window and a collection of dialogs and widgets for serving the purpose of the application. A main window provides a framework for building an application's user interface. In this chapter, we shall discuss the creation of a main window application with its predefined components and also discuss the layout management in a windowed application.

PySide provides a class named PySide.QtGui.QMainWindow derived from QWidget, and its related classes for main window management. QMainWindow has its own layout to which you can add toolbars, menu bar, status bar, dock widgets, and a central widget. The layout description of a main window is as shown in the following figure:
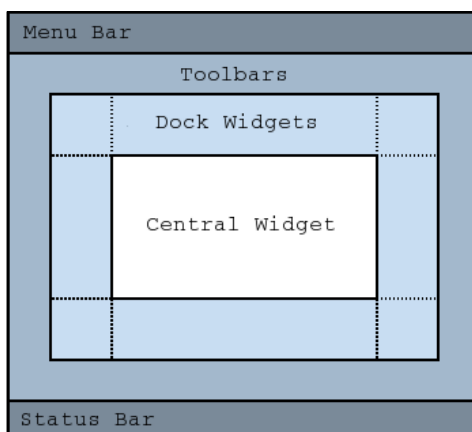
The central widget can be of any standard or custom widgets, say for example, `QTextEdit` or a `QGraphicsView`. Creating a main window without a central widget is not supported. Moving on, we will examine how to create a main window and will cover how to add its components one by one.

Creating the main window

As a first step, we will start with creating a main window by subclassing the `QMainWindow` class. The `QMainWindow` class has a constructor function similar to `QWidget` class.

> `PySide.QtGui.QMainWindow([parent=None[,flags=0]])`

The parent can be any valid `QWidget` object and flags can be a valid combination of `Qt.WindowFlags`. The following code excerpt explains how to create a main window application at a very basic level.

```python
# Import required modules
import sys, time
from PySide.QtGui import QApplication, QMainWindow

class MainWindow(QMainWindow):
  """ Our Main Window Class
  """

  def __init__(self):
    """ Constructor Fucntion
    """
    QMainWindow.__init__(self)
    self.setWindowTitle("Main Window")
    self.setGeometry(300, 250, 400, 300)

if __name__ == '__main__':
  # Exception Handling
  try:
    myApp = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

This will create a very minimal and a basic main window with no other components as shown in the following screenshot. In the forthcoming sections, we will see how to add these components in the main window:
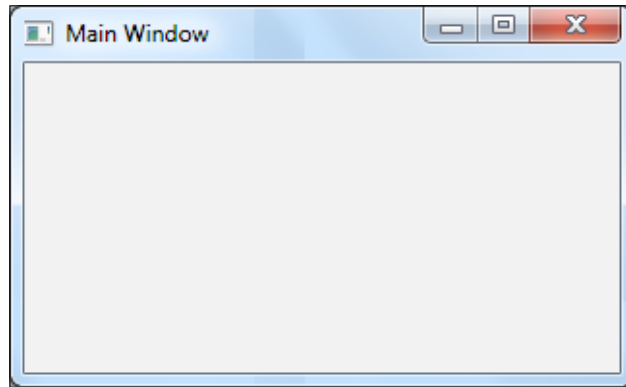
Status bar

A status bar is a horizontal information area usually found at the bottom of windows in a GUI. Its primary job is to display information about the current status of the window. A status bar can also be divided into sections, each showing different information to the users.

In PySide, a status bar can be added to the QMainWindow class by calling the function QMainWindow.setStatusBar(statusbar). It takes the object of PySide.QtGui. QStatusBar as a parameter. The properties of the status bar is defined by this class and an object of this class is returned to set a status bar. Setting this parameter to 0 will remove the status bar from the main window. A status bar can show information that can fall into any of the following three categories:

- **Temporary**: Briefly occupies most of the status bar and is mainly used to explain tool tip texts, menu entries, and so on
- **Normal**: Occupies a part of the status bar and may be temporarily hidden by temporary messages, and is used to display the current window information, page and line numbers and so on
- **Permanent**: Usually occupies a little space and is used to indicate important mode information, Caps Lock indicator, spell check info, and so on

The current status of the status bar can be retrieved by using the `QMainWindow.statusBar()` function. The temporary messages to the status bar can be set by calling the `QStatusBar.showMessage(text[,timeout=0])` function. If a timeout is set, the message will be cleared after the expiry of specified time in milliseconds. If there is no timeout, you can clear the message by calling the `QStatusBar.clearMessage()` function.

To evident this, we create a method called `CreateStatusBar()`, which has the following code snippet:

```
def CreateStatusBar(self):
  """ Function to create Status Bar
  """
  self.myStatusBar = QStatusBar()
  self.myStatusBar.showMessage('Ready', 2000)
  self.setStatusBar(self.myStatusBar)
```

Now call this function, from the main application to set a status bar on the main window. On executing this code, we could see a status bar appearing on the main window that will expire after 2 seconds, which is the timeout that we have set. If you have left the timeout option or set it as `0`, the message will appear in the status bar till another message is called on to overwrite it or till we close the application. The output window will be as given in the following screenshot.

It is also possible to set a widget in the status bar in addition to the text messages. A widget like `QProgressBar` can be added to the status bar to indicate the progress of a particular action on the main window. The `PySide.QtGui.QProgressBar` widget provides a horizontal or vertical progress bar. The progress bar is used to intimate the user an indication of the progress of an operation and to reassure them that the application is still running. We shall now see a program that implements this. Make the changes to the previous program explained as follows:
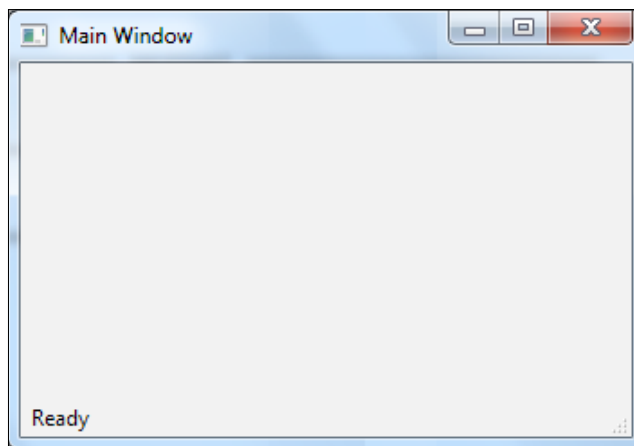
The `setMinimum(minimum)` and `setMaximum(maximum)` functions of `PySide.QtGui.QProgressBar` takes an integer value as the value to set the minimum and maximum possible step values and it will display the percentage of steps that have been completed when you later give it the current step value. The percentage is calculated by dividing the progress as follows:

```
[PySide.QtGui.QProgressBar.value() -PySide.QtGui.QProgressBar.
minimum()]
-------------------------------------------------------------------
----------
[PySide.QtGui.QProgressBar.maximum() - PySide.QtGui.QProgressBar.
minimum()]


  def __init__(self):
...
  self.setGeometry(300, 250, 400, 300)
    self.statusLabel = QLabel('Showing Progress')
    self.progressBar = QProgressBar()
    self.progressBar.setMinimum(0)
    self.progressBar.setMaximum(100)
...
```

As explained, we will now add widgets to the status bar. For this purpose, we have created two widgets namely, `self.statusLabel` and `self.progressBar` of type `QLabel` and `QProgressBar` respectively. The following code creates the status bar and add these widgets to it:

```
def CreateStatusBar(self):
    """ Function to create the status bar
    """
    self.myStatusBar = QStatusBar()
    self.progressBar.setValue(10)
    self.myStatusBar.addWidget(self.statusLabel, 1)
    self.myStatusBar.addWidget(self.progressBar, 2)
    self.setStatusBar(self.myStatusBar)
```

The function `PySide.QtGui.QStatusBar.addWidget(widget[, stretch=0])` takes two arguments. The first mandatory argument is any valid `QWidget` object that is to be added on the status bar and the second optional parameter is used to compute a suitable size for the given widget as the status bar grows and shrinks. The number defines the ratio of the status bar that the widget can use. By setting the progress bar's stretch factor to 2, we ensure that it takes two-third of the total area of the status bar.

After adding the widgets to the status bar, we write the following function to show its working state. This function will show the label as "Showing Progress" till the progress bar at the right proceeds to completion. The progress bar increments by 10 percent every time till completion after a short sleep for one second. On completion, the label changes to Ready.

```
def ShowProgress(self):
 """ Function to show progress
 """
   while(self.progressBar.value() < self.progressBar.maximum()):
     self.progressBar.setValue(self.progressBar.value() + 10)
     time.sleep(1)
   self.statusLabel.setText('Ready')
```

Now, modify the code in the main block, as shown in the following information box and execute the program. You can now see that the status bar has a working progress bar widget along with a label.

```
....
        mainWindow = MainWindow()
        mainWindow.CreateStatusBar()
        mainWindow.show()
        mainWindow.ShowProgress()
        myApp.exec_()
....
```

After making the changes, execute the program and you will see an output window as shown in the following screenshot:
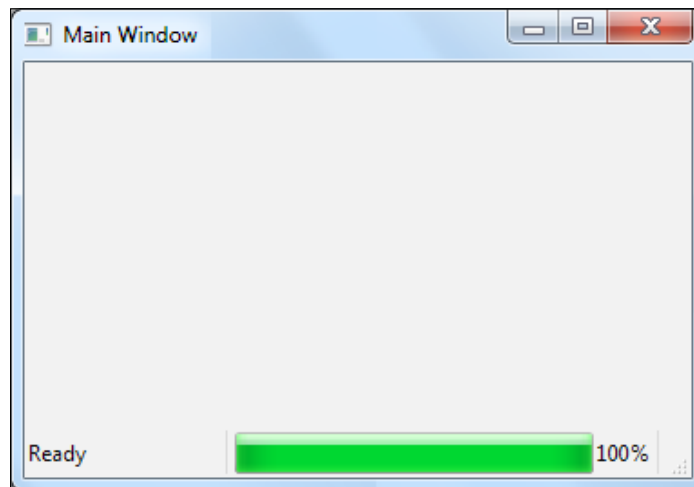
Menu bar

In the systems that use command line interface, the user may be presented with a list of commands that would be displayed as a help text to the user. The users can then choose from the list to perform their desired action. In GUI systems, this is replaced with a set of text and symbols to represent choices. By clicking on the text/symbol, the user executes the desired action. This collection is called a menu.

A menu bar is a region of a screen or application that consists of a list of pull-down menu items. A common use of menu bar is to provide convenient access to various operations such as opening a new or an existing file, save the file, print options, manipulating data, providing help window, close the application and so on. In this section, we introduce the concept of adding a menu bar to our application. Before that, we define the central widget of the application which will aid the usage of menus.

The central widget

As we have seen earlier, the central area of the `QMainWindow` can be occupied by any kind of widget. This widget can be any of the following:

- A standard `Qt` widget such as `QTextEdit` or `QGraphicsView`
- A custom widget created using primary widgets
- Plain `QWidget` with a layout manager, which acts as a parent for many other widgets
- Multiple widgets with a `QSplitter` which arranges the widgets horizontally or vertically
- MDI area which acts as a parent for other MDI windows

In our example, we use `QTextEdit` as the central widget. The `PySide.QtGui. QTextEdit` class provides a widget that is used to edit and display both plain and rich text formats. `QTextEdit` is an advanced **WYSIWYG** viewer/editor supporting rich text formatting using HTML style tags. It can display text, images, lists, and tables as well. The rich text support in `Qt` is designed to provide a fast, portable, and efficient way to add reasonable online help facilities to applications, and to provide a basis for rich text editors. The `QTextEdit` can be both used as a display widget and an editor. For our purposes, we use it as an editor.

The following piece of code sets the `QTextEdit` as the central widget. On calling this function from `main` block by using the `QApplication` object, a text editor will be set at the central space of our application. The output is shown in the screenshot following the information box containing the code excerpt:

```
def SetupComponents(self):
    """ Setting the central widget
        """
        textEdit = QTextEdit()
        self.setCentralWidget(textEdit)
```
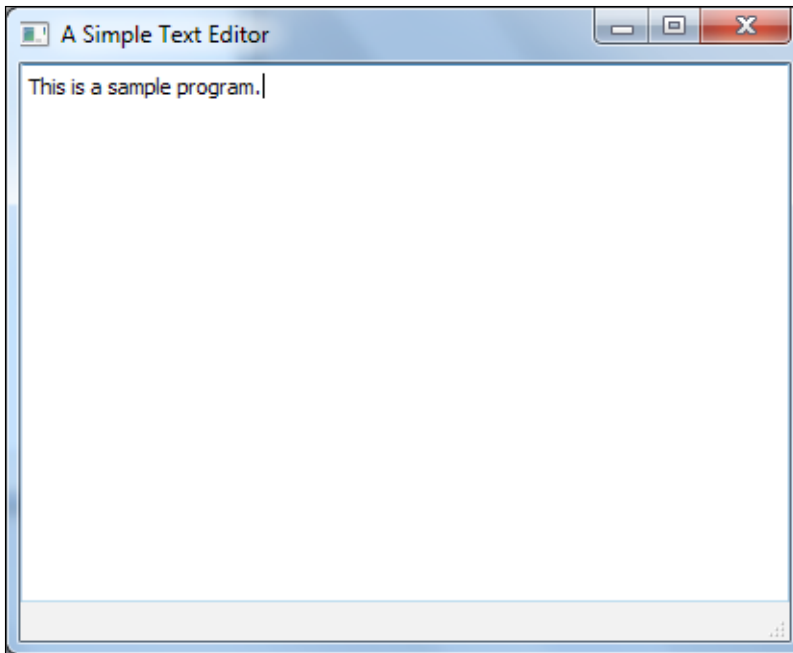

Adding a menu bar

We will now add the menu bar to our application. We use the class `PySide.QtGui. QMenuBar` for creating a menu bar. This class provides a horizontal menu bar to which the menu items can be added. You don't have to set a layout for menu bar. Different platforms use different layouts for the menu bar. In Windows system, the menu bar is usually anchored at the top of a window under the title bar. In the Macintosh system, the menu bar is usually anchored at the top of the screen. Linux systems have both these display formats depending on the GUI style. `QMenuBar` automatically sets its own geometry to the top of the parent widget and changes it appropriately whenever the parent is resized.

In the main window style applications, we use the `PySide.QtGui.QMainWindow.` `menuBar()` function to create a menu bar. This function will return an empty menu bar, which has `QMainWindow` as its parent. If you want all windows in a Mac application to share a single common menu bar, don't use this function to create it, because the menu bar created this way will have `QMainWindow` as its parent, instead you can create a menu bar with no parent by directly instantiating the `QMenuBar` class. The menu bar can also be set in the main window by using the `PySide.QtGui.QMainWindow.setMenuBar(menubar)` function, which takes a menu bar object as its parameter.

Adding menus

Once the menu bar is set, menu list can be added to it. The `PySide.QtGui.QMenu` class provides menu widget for use in menu bars, context menus, and other popup menus. A menu widget is a selection menu. It can be either a pull-down menu in a menu bar or can be a context menu. Pull-down menus are shown by the menu bar when the user clicks on the respective item or presses the specified shortcut key. Context menus are invoked by some special keyboard key or by right-clicking on it.

In the menu bar, we add menus with the function `QMenuBar.addMenu(menu)`. For the example application, we add three menus namely, `File`, `Edit`, and `About`. Inside each of these menus, we create two actions each that when triggered by click or a keyboard shortcut key combination connects to a specified slot. As defined, a menu consists of a list of action items. The `PySide.QtGui.QAction` class provides an abstract user interface action that can be inserted into widgets. Actions are common for a combination of menu items, tool bars, and keyboard shortcuts. So we create an action and attach it with different components, which are expected to perform the same functionality. Usually, when an action is created, it should be added to the relevant menu and toolbar, then connected to the slot which will perform the action. Actions are added to the menus with the `QMenu.addAction()`, `QMenu.addActions()`, and `QMenu.` `insertAction()` functions. An action is rendered vertically under the menu, and can have a text label, an optional icon drawn on the left, and a shortcut key sequence.

The following example demonstrates the creation of menu bar in the main window application. This program is a shortened version of what we will be developing as a fully working "A Simple Text Editor" application at the end of this chapter.

```
# Import required modules
import sys
from PySide.QtGui import QApplication, QMainWindow, QStatusBar, \
QTextEdit, \
    QAction, QIcon, QKeySequence
```

```python
class MainWindow(QMainWindow):
  """ Our Main Window class
  """

  def __init__(self):
    """ Constructor Function
    """
    QMainWindow.__init__(self)
    self.setWindowTitle("A Simple Text Editor")
    self.setWindowIcon(QIcon('appicon.png'))
    self.setGeometry(300, 250, 400, 300)

  def SetupComponents(self):
    """ Function to setup status bar, central widget, menu bar
    """
    self.myStatusBar = QStatusBar()
    self.setStatusBar(self.myStatusBar)
    self.myStatusBar.showMessage('Ready', 10000)
    self.textEdit = QTextEdit()
    self.setCentralWidget(self.textEdit)
    self.CreateActions()
    self.CreateMenus()
    self.fileMenu.addAction(self.newAction)
    self.fileMenu.addSeparator()
    self.fileMenu.addAction(self.exitAction)
    self.editMenu.addAction(self.copyAction)
    self.fileMenu.addSeparator()
    self.editMenu.addAction(self.pasteAction)
    self.helpMenu.addAction(self.aboutAction)


  # Slots called when the menu actions are triggered
  def newFile(self):
    self.textEdit.setText('')

  def exitFile(self):
    self.close()

  def aboutHelp(self):
    QMessageBox.about(self, "About Simple Text Editor",
        "This example demonstrates the use "
        "of Menu Bar")

  def CreateActions(self):
    """ Function to create actions for menus
    """
    self.newAction = QAction( QIcon('new.png'), '&New',
                  self, shortcut=QKeySequence.New,
                  statusTip="Create a New File",
                  triggered=self.newFile)
```

```
        self.exitAction = QAction( QIcon('exit.png'), 'E&xit',
                    self, shortcut="Ctrl+Q",
                    statusTip="Exit the Application",
                    triggered=self.exitFile)
        self.copyAction = QAction( QIcon('copy.png'), 'C&opy',
                    self, shortcut="Ctrl+C",
                    statusTip="Copy",
                    triggered=self.textEdit.copy)
        self.pasteAction = QAction( QIcon('paste.png'), '&Paste',
                    self, shortcut="Ctrl+V",
                    statusTip="Paste",
                    triggered=self.textEdit.paste)
        self.aboutAction = QAction( QIcon('about.png'), 'A&bout',
                    self, statusTip="Displays info about text editor",
                    triggered=self.aboutHelp)

    # Actual menu bar item creation
    def CreateMenus(self):
        """ Function to create actual menu bar
        """
        self.fileMenu = self.menuBar().addMenu("&File")
        self.editMenu = self.menuBar().addMenu("&Edit")
        self.helpMenu = self.menuBar().addMenu("&Help")

if __name__ == '__main__':
    # Exception Handling
    try:
        #QApplication.setStyle('plastique')
        myApp = QApplication(sys.argv)
        mainWindow = MainWindow()
        mainWindow.SetupComponents()
        mainWindow.show()
        myApp.exec_()
        sys.exit(0)
    except NameError:
        print("Name Error:", sys.exc_info()[1])
    except SystemExit:
        print("Closing Window...")
    except Exception:
        print(sys.exc_info()[1])
```
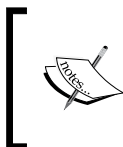
In the preceding example, the actions are created in the `CreateActions()` function. Each of the menu items are created as separate actions.

```
        self.newAction = QAction( QIcon('new.png'), '&New',
            self, shortcut=QKeySequence.New,
            statusTip="Create a New File",
            triggered=self.newFile)
```

The `New` item under `File` menu has the action instantiated as preceding code snippet. The first parameter is the Icon to be displayed on to the left of the menu item. The next parameter is the name to be displayed. The `&` represents that the letter followed by it should be underlined, and can be accessed by pressing the *Alt* + <letter> combination as in many other window applications. The third parameter implies the parent which is the main window here. The fourth and fifth keyword parameters represent the shortcut key combination for easy access of the menu item and status tip to be displayed on highlighting the menu item respectively. The shortcut key can be either a predefined combination by the `PySide.QtGui.QKeySequence` class or a valid combination as given by the user. The `PySide.QtGui.QKeySequence` class encapsulates a key sequence as used by shortcuts. It defines a combination of keys that can be used by different operating system platforms. The last parameter defines the slot to be called when the menu item is triggered, which can be a logical group of code executing the desired functionality. We define the other actions similar to this.

Once the actions are created, we add these actions to the menu items. The top level menu bar is created in the `CreateMenus()` function and the actions of the menu items are added as follows:

```
self.fileMenu.addAction(self.newAction)
```

The menu items can be grouped by adding a separator between the items which is done as given in the following code line:

```
self.fileMenu.addSeparator()
```

Execute the program and witness the working of menus.

Tool bar

A tool bar is a panel of icons associated with actions that are available for easy access of menus. In PySide, the tool bars are implemented in the `PySide.QtGui.QToolBar` class. The toolbar is added to the main window with `addToolBar()` function. The tool bar is initially positioned at the top of the window below the menu bar. This can be adjusted with the `QToolBar.setAllowedAreas()` function. The tool bar can be set movable or immovable by setting it with the `QToolBar.setMovable()` function. The style and size of the icons can be defined by the underlying platform, which could also be controlled. When it is resized in a way too small that can hold all the icons, an extension button will appear which on click expands to all items.

Toolbar buttons are added by adding actions as seen in the menu bar creation in the previous section. The same actions can be used in the toolbars too. The following example demonstrates the creation of toolbar and its usage:

```
def CreateToolBar(self):
    """ Function to create tool bar
    """
    self.mainToolBar = self.addToolBar('Main')
```

The actions can be added as follows. These lines should be appended in the `SetupComponents()` function after invoking the `CreateToolBar()` module in our previous example:

```
self.mainToolBar.addAction(self.newAction)
self.mainToolBar.addSeparator()
self.mainToolBar.addAction(self.copyAction)
self.mainToolBar.addAction(self.pasteAction)
```

Thus, we have seen how to add the discussed components in the main window.

Layout management

A layout can be defined as the sizing, spacing, and placement of content within a GUI window. It defines the visual structure of a user interface. Effective layout management would please users by assisting them in locating quickly what the users wants most from the application, and also helps in making out the differences between a good informative design and a confusing, puzzled designs of the application. Therefore, management of layout in a window style application is a crucial success factor for any GUI application. A good layout must have a priority focus, smooth flow, logical grouping, relative emphasis, and coordinated alignment.

In PySide, we follow two approaches to layout management. They are as follows:

- **Absolute positioning**: A crude way of setting the layout manually for each widget by giving them their position and size
- **Layout containers**: A way to handle automatic positioning and resize of widgets by the layout management classes used in Qt

Absolute positioning

Absolute positioning is the most crude and naive form of arranging widgets in the window. This is achieved by giving a hard-wired position and size to all the widgets in the window. Usually, we use the widget's `move(x, y)` function where *x* and *y* are the horizontal and vertical distance, respectively, from the top left corner of the form to the top left corner of the widget. We have already seen this method of layout positioning in our previous chapters in positioning the `About` and `Quit` buttons.

This method of layout management is highly discouraged and not effective due to the following disadvantages:

- Calculating the size and position for each widget manually is a really cumbersome task
- Resizing the window changes the layout
- It do not respond to style changes
- Internationalization and font changes becomes very difficult as the label text may overflow or underflow
- May appear completely different in different resolutions

Therefore, it is highly advised to use the layout containers for layout management. We have already seen examples for this type of layout management.

Layout containers

All widgets can use layouts to manage their children. If these are used in a proper way, the following functions are achieved:

- Sensible default size for windows
- Positioning of child widgets
- Resize handling
- Automatic updates when contents change including font size, hiding, removal of widgets, and so on

The container class alienates all the disadvantages discussed in absolute positioning and is the most widely used layout management system. They are more flexible and adjusts the layout in accordance with the style of the different platform. We will look into the commonly used layout containers inherited from the `PySide.QtGui.QLayout` class.

- `QBoxLayout`: It lines up widgets horizontally or vertically
    - `QHBoxLayout`: It lines up widgets horizontally
    - `QVBoxLayout`: It lines up widgets vertically
- `QGridLayout`: Lays out widgets in a grid
- `QFormLayout`: Manages form of input widgets and their label
- `QStackedLayout`: Stack of widgets where only one widget is visible at a time

We will discuss each of them in little detail.

QBoxLayout

The `PySide.QtGui.QBoxLayout` class takes the space it gets, divides it up into row of boxes and makes each managed widgets fill one box. The orientation or direction of the box layout can be either vertical (column wise) or horizontal (row wise). The direction can take any one of the following values:

- `QBoxLayout.LeftToRight`: It takes horizontal direction from left to right
- `QBoxLayout.RightToLeft`: It takes horizontal direction from right to left
- `QBoxLayout.TopToBottom`: It takes vertical direction from top to bottom
- `QBoxLayout.BottomToTop`: It takes vertical direction from bottom to top

Each of the filled widgets will get its minimum size at the least and its maximum size at the most. The extra space is shared between the widgets according to their stretch factor. The `QBoxLayout` class can be attached to any of the parent layout as it is not the top level layout.

The widgets can be added to the `QBoxLayout` function using following four functions:

- `QBoxLayout.addWidget()`: It helps to add widgets
- `QBoxLayout.addSpacing()`: It creates an empty box for spacing
- `QBoxLayout.addStretch()`: It creates an empty stretchable box
- `QBoxLayout.addLayout()`: It adds a box with another layout and define its stretch factor

The add functions can be replaced with insert functions, for example, `QBoxLayout.insertWidget()` to insert a box at a specified position in the layout.

This class also includes two margins. The `Pyside.QtGui.QLayout.setContentsMargins()` function sets the width of the outer border on each side of the widget and `PySide.QtGui.QBoxLayout.setSpacing()` sets the width between the neighboring boxes. The default margin spaces differ by the application style.

The most convenient and easy form to use the `QBoxLayout` class is to instantiate one of its inherited classes, `QVBoxLayout` and `QHBoxLayout` for vertical and horizontal direction layouts respectively. The following programs will explain the usage of `QVBoxLayout` and `QHBoxLayout`.

QHBoxLayout

An example for horizontal layout is as follows:

```
class MainWindow(QWidget):
  """ Our Main Window class
  """
  def __init__(self):
    """ Constructor Function
    """
    QWidget.__init__(self)
    self.setWindowTitle("Horizontal Layout")
    self.setGeometry(300, 250, 400, 300)

  def SetLayout(self):
    """ Function to add buttons and set the layout
    """
    horizontalLayout = QHBoxLayout(self)
    hButton1 = QPushButton('Button 1', self)
    hButton2 = QPushButton('Button 2', self)
    hButton3 = QPushButton('Button 3', self)
    hButton4 = QPushButton('Button 4', self)
    horizontalLayout.addWidget(hButton1)
    horizontalLayout.addWidget(hButton2)
    horizontalLayout.addWidget(hButton3)
    horizontalLayout.addWidget(hButton4)

    self.setLayout(horizontalLayout)
```

QVBoxLayout

An example for vertical layout is as follows:

```
  def SetLayout(self):
    verticalLayout = QVBoxLayout(self)
    vButton1 = QPushButton('Button 1', self)
    vButton2 = QPushButton('Button 2', self)
    vButton3 = QPushButton('Button 3', self)
    vButton4 = QPushButton('Button 4', self)

    verticalLayout.addWidget(vButton1)
    verticalLayout.addWidget(vButton2)
    verticalLayout.addStretch()
    verticalLayout.addWidget(vButton3)
    verticalLayout.addWidget(vButton4)

    self.setLayout(verticalLayout)
```
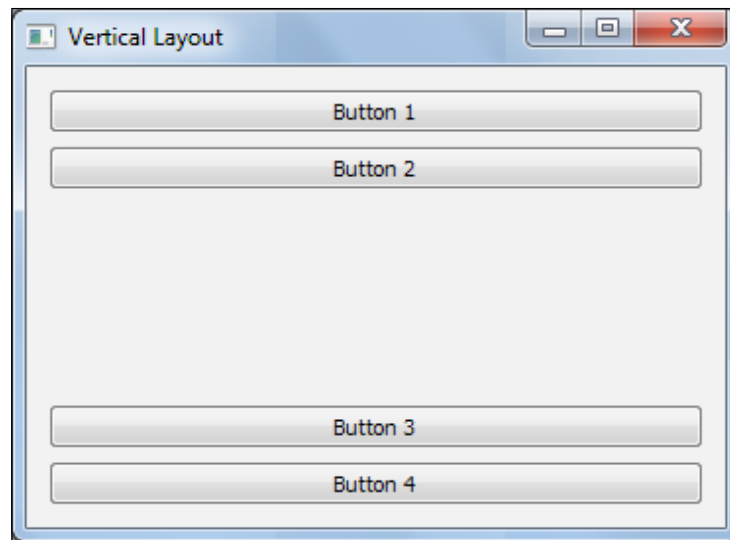
QGridLayout

The `PySide.QtGui.QGridLayout` class takes the space available to it and divides up into rows and columns and puts each widget it manages into the correct cell. Each row/column has a minimum width and a stretch factor. The widgets are added into the grid layout using the `addWidget()` function and the layout puts it into the correct cell. It is also possible for a widget to span across multiple rows/columns. The `addItem()` and `addLayout()` methods can also be used to insert widgets or other layouts into it.

The grid layout also includes two margins as discussed in the box layout. An example program for the usage of grid layout is as follows.

```
def SetLayout(self):
    gridLayout = QGridLayout(self)
    gButton1 = QPushButton('Button 1', self)
    gButton2 = QPushButton('Button 2', self)
    gButton3 = QPushButton('Button 3', self)
    gButton4 = QPushButton('Button 4', self)
    gButton5 = QPushButton('Button 5', self)
    gridLayout.addWidget(gButton1, 0, 0)
    gridLayout.addWidget(gButton2, 0, 1)
    gridLayout.addWidget(gButton3, 1, 0, 1, 2)
    gridLayout.addWidget(gButton4, 2, 0)
    gridLayout.addWidget(gButton5, 2, 1)
    self.setLayout(gridLayout)
```

QFormLayout

The `PySide.QtGui.QFormLayout` class is a higher level alternative to basic forms of layout classes, which lays out its widgets in a two column form. Usually, the left column consists of `label` and the right column consists of `field` widgets such as line editors, combo box, spin box, and so on.

The form layout can be achieved through grid layout but using it directly on the form like widgets have the following advantages:

- Adherence to different platform's look and feel
- Support for wrapping long rows, using *RowWrapPolicy* control
- API for creating *label-field* pairs
- Support for expanding fields, using *FieldGrowthPolicy* control

The spacing between the rows of forms can be set using the `setHorizontalSpacing()` and `setVerticalSpacing()` functions of this class. An example for the form layout is as follows:

```
def SetLayout(self):
    formLayout = QFormLayout(self)
    labelUsername = QLabel("Username")
    txtUsername = QLineEdit()
    labelPassword = QLabel("Password")
    txtPassword = QLineEdit()
    formLayout.addRow(labelUsername, txtUsername)
    formLayout.addRow(labelPassword, txtPassword)
    self.setLayout(formLayout)
```

QStackedLayout

The `PySide.QtGui.QStackedLayout` class lays out a set of child widgets and shows only one at a time, hiding the others from the user. The layout can be initially populated with a number of child widgets and later on any one of them can be selected to be shown to the user depending on the choice in window. There is no intrinsic way given by the layout itself for the users to select a widget from the available child widgets. This is achieved through using either `QComboBox` or `QListWidget` widgets. On populating the layout, the child widgets are added to an internal list and the index is returned by the layout to select child widgets. The widgets can be inserted and removed using the `insertWidget()` and `removeWidget()` functions respectively.

The implementation of stack layout is left as an exercise for you. The sample program can be found in the code samples that come along with this book.

SDI and MDI

In many GUI applications, we would arrive at a situation to open more than one document at a time for processing. We would want to design our application to handle this. This can be achieved by either of the two approaches namely, SDI and MDI. A **Single Document Interface** or **SDI** application implements this by creating separate windows for each of the documents. This is done by creating a window subclass that handles everything by itself, including loading, saving, and clean-up, and so on. Each of the documents will be a clone of the main window having a separate menu bar, toolbar, and status bar on its own. Each of the main window instances must be able to act independently. However, there are some disadvantages to this approach. This approach would consume a lot of resources and would be very inconvenient to open many windows at a time and keep track of them.

The second approach is to use a **Multiple Document Interface** or **MDI** application where the central widget is instantiated with multiple instances. All these widgets will be interrelated within the main window and shares the common menu bar, toolbar, and other components. MDI application will use lesser resources as compared to SDI applications. The MDI applications are provided with an extra menu to manage between windows, as shifting between them is not controlled by the underlying operating system. We will discuss more about SDI and MDI, and its implementation in *Chapter 5*, *Dialogs and Widgets*.

A simple text editor

The following is the implementation of a simple text editor extended from the previous examples in its building version. This code contains some new features such as, QFontDialog, QFileDialog, and so on; which are discussed in *Chapter 5*, *Dialogs and Widgets*. Otherwise, the following code is self explanatory:

```python
# Import required modules
import sys
from PySide.QtGui import *

class MainWindow(QMainWindow):
    """ Our Main Window class
    """

    def __init__(self, fileName=None):
        """ Constructor Function
        """
```

```
        QMainWindow.__init__(self)
        self.setWindowTitle("A Simple Text Editor")
        self.setWindowIcon(QIcon('appicon.png'))
        self.setGeometry(100, 100, 800, 600)

        self.textEdit = QTextEdit()
        self.setCentralWidget(self.textEdit)
        self.fileName = None

        self.filters = "Text files (*.txt)"

    def SetupComponents(self):
        """ Function to setup status bar, central widget, menu
          bar, tool bar
        """
        self.myStatusBar = QStatusBar()
        self.setStatusBar(self.myStatusBar)
        self.myStatusBar.showMessage('Ready', 10000)

        self.CreateActions()
        self.CreateMenus()
        self.CreateToolBar()
        self.fileMenu.addAction(self.newAction)
        self.fileMenu.addAction(self.openAction)
        self.fileMenu.addAction(self.saveAction)
        self.fileMenu.addSeparator()
        self.fileMenu.addAction(self.exitAction)
        self.editMenu.addAction(self.cutAction)
        self.editMenu.addAction(self.copyAction)
        self.editMenu.addAction(self.pasteAction)
        self.editMenu.addSeparator()
        self.editMenu.addAction(self.undoAction)
        self.editMenu.addAction(self.redoAction)
        self.editMenu.addSeparator()
        self.editMenu.addAction(self.selectAllAction)
        self.formatMenu.addAction(self.fontAction)
        self.helpMenu.addAction(self.aboutAction)
        self.helpMenu.addSeparator()
        self.helpMenu.addAction(self.aboutQtAction)
        self.mainToolBar.addAction(self.newAction)
        self.mainToolBar.addAction(self.openAction)
        self.mainToolBar.addAction(self.saveAction)
        self.mainToolBar.addSeparator()
```

```python
    self.mainToolBar.addAction(self.cutAction)
    self.mainToolBar.addAction(self.copyAction)
    self.mainToolBar.addAction(self.pasteAction)
    self.mainToolBar.addSeparator()
    self.mainToolBar.addAction(self.undoAction)
    self.mainToolBar.addAction(self.redoAction)


# Slots called when the menu actions are triggered
def newFile(self):
  self.textEdit.setText('')
  self.fileName = None

def openFile(self):
  self.fileName, self.filterName = \
    QFileDialog.getOpenFileName(self)
  self.textEdit.setText(open(self.fileName).read())

def saveFile(self):
  if self.fileName == None or self.fileName == '':
    self.fileName, self.filterName = \
      QFileDialog.getSaveFileName(self, \
      filter=self.filters)
  if(self.fileName != ''):
    file = open(self.fileName, 'w')
    file.write(self.textEdit.toPlainText())
    self.statusBar().showMessage("File saved", 2000)

def exitFile(self):
  self.close()

def fontChange(self):
  (font, ok) = QFontDialog.getFont(QFont("Helvetica
    [Cronyx]", 10), self)
  if ok:
    self.textEdit.setCurrentFont(font)

def aboutHelp(self):
  QMessageBox.about(self, "About Simple Text Editor",
      "A Simple Text Editor where you can edit and save
        files")
```

```
def CreateActions(self):
  """ Function to create actions for menus
  """
  self.newAction = QAction( QIcon('new.png'), '&New',
            self, shortcut=QKeySequence.New,
            statusTip="Create a New File",
            triggered=self.newFile)
  self.openAction = QAction( QIcon('open.png'), 'O&pen',
            self, shortcut=QKeySequence.Open,
            statusTip="Open an existing file",
            triggered=self.openFile)
  self.saveAction = QAction( QIcon('save.png'), '&Save',
            self, shortcut=QKeySequence.Save,
            statusTip="Save the current file to
              disk",
            triggered=self.saveFile)
  self.exitAction = QAction( QIcon('exit.png'), 'E&xit',
            self, shortcut="Ctrl+Q",
            statusTip="Exit the Application",
            triggered=self.exitFile)
  self.cutAction = QAction( QIcon('cut.png'), 'C&ut',
            self, shortcut=QKeySequence.Cut,
            statusTip="Cut the current selection to clipboard",
            triggered=self.textEdit.cut)
  self.copyAction = QAction( QIcon('copy.png'), 'C&opy',
            self, shortcut=QKeySequence.Copy,
            statusTip="Copy the current selection to clipboard",
            triggered=self.textEdit.copy)
  self.pasteAction = QAction( QIcon('paste.png'), '&Paste',
            self, shortcut=QKeySequence.Paste,
            statusTip="Paste the clipboard's content in current
              location",
            triggered=self.textEdit.paste)
  self.selectAllAction = QAction( QIcon('selectAll.png'),
    'Select All',
            self, statusTip="Select All",
            triggered=self.textEdit.selectAll)
  self.redoAction = QAction( QIcon('redo.png'),'Redo', self,
            shortcut=QKeySequence.Redo,
            statusTip="Redo previous action",
            triggered=self.textEdit.redo)
  self.undoAction = QAction( QIcon('undo.png'),'Undo', self,
            shortcut=QKeySequence.Undo,
            statusTip="Undo previous action",
```

```python
                triggered=self.textEdit.undo)
    self.fontAction = QAction( 'F&ont', self,
              statusTip = "Modify font properties",
              triggered = self.fontChange)
    self.aboutAction = QAction( QIcon('about.png'), 'A&bout',
              self, statusTip="Displays info about text editor",
              triggered=self.aboutHelp)
    self.aboutQtAction = QAction("About &Qt", self,
              statusTip="Show the Qt library's About box",
              triggered=qApp.aboutQt)


def CreateMenus(self):
    self.fileMenu = self.menuBar().addMenu("&File")
    self.editMenu = self.menuBar().addMenu("&Edit")
    self.formatMenu = self.menuBar().addMenu("F&ormat")
    self.helpMenu = self.menuBar().addMenu("&Help")


def CreateToolBar(self):
    self.mainToolBar = self.addToolBar('Main')


if __name__ == '__main__':
  # Exception Handling
  try:
    myApp = QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.SetupComponents()
    mainWindow.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

The sample output of the preceding program is given in the following screenshot:
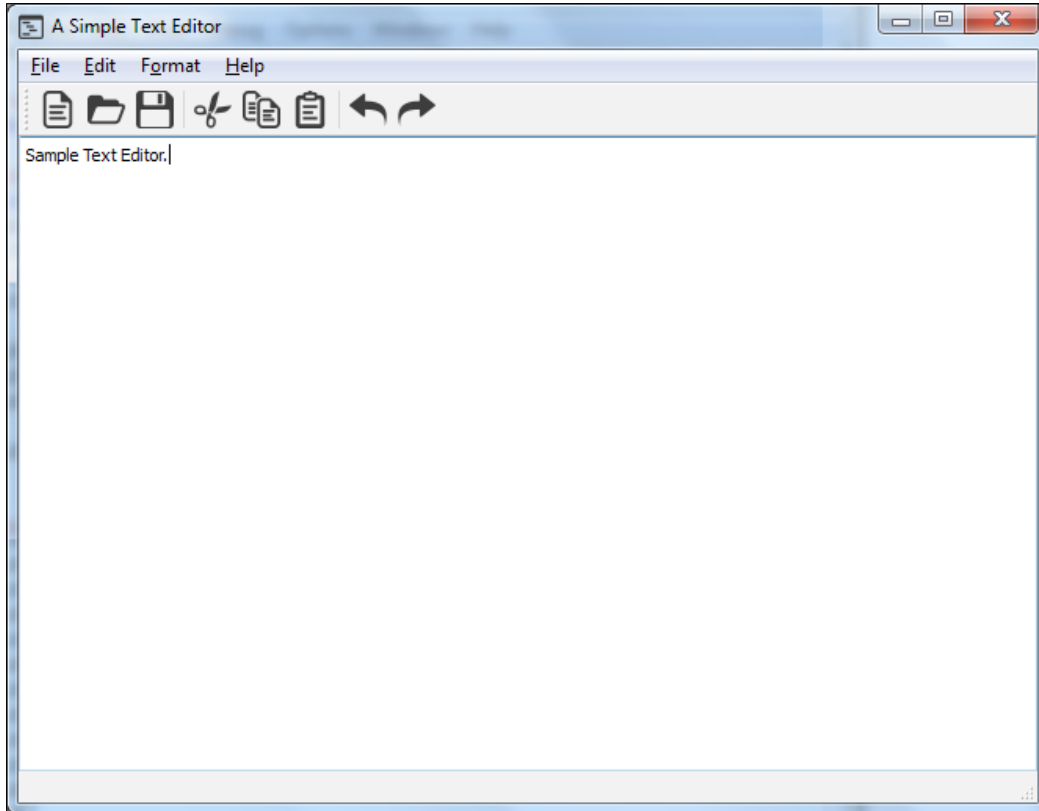
The preceding example is a very simple text editor performing some basic functions. We may build upon this example in the coming chapters as and when we discuss some new features.

Summary

In this chapter, we have seen how to create the most widely used main window styled applications. We have also learned about the layout management in applications. We completed the chapter with a real time text editor example. We may build on this example in coming chapters.

4
Events and Signals

In the chapters that we have seen so far, we tried out various implementations of readily available functions are extended by the Qt objects. In this chapter, we will look into some of the internal implementation working concepts of those functions. Being an event-driven toolkit, events and event delivery play an important role in the Qt architecture. We will start this chapter by discussing events and signals, their implementation, and will go on to discuss handling drag-and-drop events, and drawing functionalities.

Event management

An event in Qt is an object inherited from the abstract `QEvent` class which is a notification of something significant that has happened. Events become more useful in creating custom widgets on our own. An event can happen either within an application or as a result of an outside activity that the application needs to know about. When an event occurs, Qt creates an event object and notifies to the instance of an `QObject` class or one of its subclasses through their `event()` function. Events can be generated from both inside and outside the application. For instance, the `QKeyEvent` and `QMouseEvent` object represent some kind of keyboard and mouse interaction and they come from the window manager; the `QTimerEvent` objects are sent to `QObject` when one of its timers fires, and they usually come from the operating system; the `QChildEvent` objects are sent to `QObject` when a child is added or removed and they come from inside of your Qt application.

The users of PySide usually get confused with events and signals. Events and signals are two parallel mechanisms used to accomplish the same thing. As a general difference, signals are useful when using a widget, whereas events are useful when implementing the widget. For example, when we are using a widget like `QPushButton`, we are more interested in its `clicked()` signal than in the low-level mouse press or key press events that caused the signal to be emitted. But if we are implementing the `QPushButton` class, we are more interested in the implementation of code for mouse and key events. Also, we usually handle events but get notified by signal emissions.

Event loop

All the events in Qt will go through an event loop. One main key concept to be noted here is that the events are not delivered as soon as they are generated; instead they're queued up in an event queue and processed later one-by-one. The event dispatcher will loop through this queue and dispatch these events to the target `QObject` and hence it is called an **event loop**. Qt's main event loop dispatcher, `QCoreApplication.exec()` will fetch the native window system events from the event queue and will process them, convert them into the `QEvent` objects, and send it to their respective target `QObject`.

A simple event loop can be explained as described in the following pseudocode:

```
while(application_is_active)
{
  while(event_exists_in_event_queue)
    process_next_event();

  wait_for_more_events();
}
```

The Qt's main event loop starts with the `QCoreApplication::exec()` call and this gets blocked until `QCoreApplication::exit()` or `QCoreApplication::quit()` is called to terminate the loop. The `wait_for_more_events()` function blocks until some event is generated. This blocking is not a busy wait blocking and will not burn the CPU resources. Generally the event loop can be awaken by a window manager activity, socket activity, timers, or event posted by other threads. All these activities require a running event loop. It is more important not to block the event loop because when it is struck, widgets will not update themselves, timers won't fire, networking communications will slow down and stop. In short, your application will not respond to any external or internal events and hence it is advised to quickly react to events and return to the event loop as soon as possible.

Event processing

Qt offers five methods to do event processing. They are:

- By re-implementing a specific event handler like `keyPressEvent()`, `paintEvent()`

- By re-implementing the `QObject::event()` class

- Installing an event filter on a single QObject

- Installing an event filter on the QApplication object

- Subclassing QApplication and re-implementing `notify()`

Generally, this can be broadly divided into re-implementing event handlers and installing event filters. We will see each of them in little detail.

Reimplementing event handlers

We can implement the task at hand or control a widget by reimplementing the virtual event handling functions. The following example will explain how to reimplement a few most commonly used events, a key press event, a mouse double-click event, and a window resize event. We will have a look at the code first and defer the explanation after the code:

```python
# Import necessary modules
import sys
from PySide.QtGui import *
from PySide.QtCore import *

# Our main widget class
class MyWidget(QWidget):
  # Constructor function
  def __init__(self):
    QWidget.__init__(self)
    self.setWindowTitle("Reimplementing Events")
    self.setGeometry(300, 250, 300, 100)

    self.myLayout = QVBoxLayout()
    self.myLabel = QLabel("Press 'Esc' to close this App")
    self.infoLabel = QLabel()
    self.myLabel.setAlignment(Qt.AlignCenter)
    self.infoLabel.setAlignment(Qt.AlignCenter)
    self.myLayout.addWidget(self.myLabel)
    self.myLayout.addWidget(self.infoLabel)
    self.setLayout(self.myLayout)
```

```
# Function reimplementing Key Press, Mouse Click and Resize Events
def keyPressEvent(self, event):
  if event.key() == Qt.Key_Escape:
    self.close()

def mouseDoubleClickEvent(self, event):
  self.close()

def resizeEvent(self, event):
  self.infoLabel.setText("Window Resized to QSize(%d, %d)" % (event.
size().width(), event.size().height()))

if __name__ =='__main__':
  # Exception Handling
  try:
    myApp = QApplication(sys.argv)
    myWidget = MyWidget()
    myWidget.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

In the preceding code, the `keyPressEvent()` function reimplements the event
generated as a result of pressing a key. We have implemented in such a way that the
application closes when the *Esc* key is pressed. On running this code, we would get a
output similar to the one shown in the following screenshot:
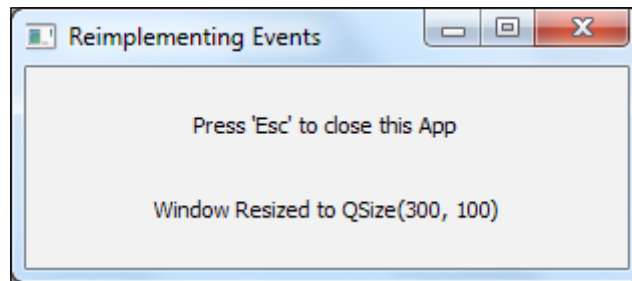
The application will be closed if you press the *Esc* key. The same functionality is
implemented on a mouse double-click event. The third event is a resize event. This
event gets triggered when you try to resize the widget. The second line of text in
the window will show the size of the window in (width, height) format. You could
witness the same on resizing the window.

Similar to `keyPressEvent()`, we could also implement `keyReleaseEvent()` that would be triggered on release of the key. Normally, we are not very interested in the key release events except for the keys where it is important. The specific keys where the release event holds importance are the modifier keys such as *Ctrl*, *Shift*, and *Alt*. These keys are called modifier keys and can be accessed using `QKeyEvent::modifiers`. For example, the key press of a *Ctrl* key can be checked using `Qt.ControlModifier`. The other modifiers are `Qt.ShiftModifier` and `Qt.AltModifier`. For instance, if we want to check the press event of combination of *Ctrl* + *PageDown* key, we could have the check as:

```
if event.key() == Qt.Key_PageDown and
event.modifiers() == Qt.ControlModifier:
  print("Ctrl+PgDn Key is pressed")
```

Before any particular key press or mouse click event handler function, say, for example, `keyPressEvent()` is called, the widget's `event()` function is called first. The `event()` method may handle the event itself or may delegate the work to a specific event handler like `resizeEvent()` or `keyPressEvent()`. The implementation of the `event()` function is very helpful in some special cases like the *Tab* key press event. In most cases, the widget with the keyboard focuses the `event()` method will call `setFocus()` on the next widget in the tab order and will not pass the event to any of the specific handlers. So we might have to re-implement any specific functionality for the *Tab* key press event in the `event()` function. This behavior of propagating the key press events is the outcome of Qt's Parent-Child hierarchy. The event gets propagated to its parent or its grand-parent and so on if it is not handled at any particular level. If the top-level widget also doesn't handle the event it is safely ignored. The following code shows an example for reimplementing the `event()` function:

```
class MyWidget(QWidget):
  # Constructor function
  def __init__(self):
    QWidget.__init__(self)
    self.setWindowTitle("Reimplementing Events")
    self.setGeometry(300, 250, 300, 100)
    self.myLayout = QVBoxLayout()
    self.myLabel1 = QLabel("Text 1")
    self.myLineEdit1 = QLineEdit()
    self.myLabel2 = QLabel("Text 2")
    self.myLineEdit2 = QLineEdit()
    self.myLabel3 = QLabel("Text 3")
    self.myLineEdit3 = QLineEdit()
    self.myLayout.addWidget(self.myLabel1)
    self.myLayout.addWidget(self.myLineEdit1)
```

```
        self.myLayout.addWidget(self.myLabel2)
        self.myLayout.addWidget(self.myLineEdit2)
        self.myLayout.addWidget(self.myLabel3)
        self.myLayout.addWidget(self.myLineEdit3)
        self.setLayout(self.myLayout)

    # Function reimplementing event() function
    def event(self, event):
        if event.type()== QEvent.KeyRelease and event.key()== Qt.Key_Tab:
            self.myLineEdit3.setFocus()
            return True
        return QWidget.event(self,event)
```

In the preceding example, we try to mask the default behavior of the *Tab* key. If you haven't implemented the `event()` function, pressing the *Tab* key would have set focus to the next available input widget. You will not be able to detect the *Tab* key press in the `keyPress()` function as described in the previous examples, since the key press is never passed to them. Instead, we have to implement it in the `event()` function. If you execute the preceding code, you would see that every time you press the *Tab* key the focus will be set into the third `QLineEdit` widget of the application. Inside the `event()` function, it is more important to return the value from the function. If we have processed the required operation, True is returned to indicate that the event is handled successfully, else, we pass the event handling to the parent class's `event()` function.

Installing event filters

One of the interesting and notable features of Qt's event model is to allow a `QObject` instance to monitor the events of another `QObject` instance before the latter object is even notified of it. This feature is very useful in constructing custom widgets comprising of various widgets altogether. Consider that you have a requirement to implement a feature in an internal application for a customer such that pressing the *Enter* key must have to shift the focus to next input widget. One way to approach the problem is to reimplement the `keyPressEvent()` function for all the widgets present in the custom widget. Instead, this can be achieved by reimplementing the `eventFilter()` function for the custom widget. If we implement this, the events will first be passed on to the custom widget's `eventFilter()` function before being passed on to the target widget. An example is implemented as follows:

```
    def eventFilter(self, receiver, event):
        if(event.type() == QEvent.MouseButtonPress):
            QMessageBox.information(None,"Filtered Mouse Press Event!!",
                'Mouse Press Detected')
            return True
        return super(MyWidget,self).eventFilter(receiver, event)
```

Remember to return the result of event handling, or pass it on to the parent's `eventFilter()` function. To invoke `eventFilter()`, it has to be registered as follows in the constructor function:

```
self.installEventFilter(self)
```

The event filters can also be implemented for the `QApplication` as a whole. This is left as an exercise for you to discover.

Reimplementing the notify() function

The final way of handling events is to reimplement the `notify()` function of the `QApplication` class. This is the only way to get all the events before any of the event filters discussed previously are notified. The event gets notified to this function first before it gets passed on to the event filters and specific event functions. The use of `notify()` and other event filters are generally discouraged unless it is absolutely necessary to implement them because handling them at top level might introduce unwanted results, and we might end up in handling the events that we don't want to. Instead, use the specific event functions to handle events. The following code excerpt shows an example of re-implementing the `notify()` function:

```
class MyApplication(QApplication):
  def __init__(self, args):
  super(MyApplication, self).__init__(args)

  def notify(self, receiver, event):
    if (event.type() == QEvent.KeyPress):
      QMessageBox.information(None, "Received Key Release EVent", "You
Pressed: "+ event.text())
    return super(MyApplication, self).notify(receiver, event)
```

Signals and slots

The fundamental part of any GUI program is the communication between the objects. Signals and slots provide a mechanism to define this communication between the actions happened and the result proposed for the respective action. Prior to Qt's modern implementation of signal/slot mechanism, older toolkits achieve this kind of communication through callbacks. A callback is a pointer to a function, so if you want a processing function to notify about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback whenever appropriate. This mechanism does not prove useful in the later advancements due to some flaws in the callback implementation.

A signal is an observable event, or at least notification that the event has happened. A slot is a potential observer, more usually a function that is called. In order to establish communication between them, we connect a signal to a slot to establish the desired action. We have already seen the concept of connecting a signal to a slot in the earlier chapters while designing the text editor application. Those implementations handle and connect different signals to different objects. However, we may have different combinations as defined in the bullet points:

- One signal can be connected to many slots
- Many signals can be connected to the same slot
- A signal can be connected to other signals
- Connections can be removed

PySide offers various predefined signals and slots such that we can connect a predefined signal to a predefined slot and do nothing else to achieve what we want. However, it is also possible to define our own signals and slots. Whenever a signal is emitted, Qt will simply throw it away. We can define the slot to catch and notice the signal that is being emitted. The first code excerpt that follows this text will be an example for connecting predefined signals to predefined slots and the latter will discuss the custom user defined signals and slots.

The first example is a simple EMI calculator application that takes the Loan Amount, Rate of Interest, and Number of Years as its input, and calculates the EMI per month and displays it to the user. To start with, we set in a layout the components required for the EMI calculator application. The Amount will be a text input from the user. The rate of years will be taken from a spin box input or a dial input. A spin box is a GUI component which has its minimum and maximum value set, and the value can be modified using the up and down arrow buttons present at its side. The dial represents a clock like widget whose values can be changed by dragging the arrow. The Number of Years value is taken by a spin box input or a slider input:

```python
class MyWidget(QWidget):
  def __init__(self):
    QWidget.__init__(self)
    self.amtLabel = QLabel('Loan Amount')
    self.roiLabel = QLabel('Rate of Interest')
    self.yrsLabel = QLabel('No. of Years')
    self.emiLabel = QLabel('EMI per month')
    self.emiValue = QLCDNumber()

    self.emiValue.setSegmentStyle(QLCDNumber.Flat)
    self.emiValue.setFixedSize(QSize(130,30))
    self.emiValue.setDigitCount(8)
```

```
        self.amtText = QLineEdit('10000')
        self.roiSpin = QSpinBox()
        self.roiSpin.setMinimum(1)
        self.roiSpin.setMaximum(15)
        self.yrsSpin = QSpinBox()
        self.yrsSpin.setMinimum(1)
        self.yrsSpin.setMaximum(20)

        self.roiDial = QDial()
        self.roiDial.setNotchesVisible(True)
        self.roiDial.setMaximum(15)
        self.roiDial.setMinimum(1)
        self.roiDial.setValue(1)
        self.yrsSlide = QSlider(Qt.Horizontal)
        self.yrsSlide.setMaximum(20)
        self.yrsSlide.setMinimum(1)

        self.calculateButton = QPushButton('Calculate EMI')


        self.myGridLayout = QGridLayout()

        self.myGridLayout.addWidget(self.amtLabel, 0, 0)
        self.myGridLayout.addWidget(self.roiLabel, 1, 0)
        self.myGridLayout.addWidget(self.yrsLabel, 2, 0)
        self.myGridLayout.addWidget(self.amtText, 0, 1)
        self.myGridLayout.addWidget(self.roiSpin, 1, 1)
        self.myGridLayout.addWidget(self.yrsSpin, 2, 1)
        self.myGridLayout.addWidget(self.roiDial, 1, 2)
        self.myGridLayout.addWidget(self.yrsSlide, 2, 2)
        self.myGridLayout.addWidget(self.calculateButton, 3, 1)

        self.setLayout(self.myGridLayout)
        self.setWindowTitle("A simple EMI calculator")
```

Until now, we have set the components that are required for the application. Note that, the application layout uses a grid layout option. The next set of code is also defined in the contructor's __init__ function of the `MyWidget` class which will connect the different signals to slots. There are different ways by which you can use a connect function. The code explains the various options available:

```
        self.roiDial.valueChanged.connect(self.roiSpin.setValue)
        self.connect(self.roiSpin, SIGNAL("valueChanged(int)"), self.
    roiDial.setValue)
```

In the first line of the previous code, we connect the `valueChanged()` signal of roiDial to call the slot of roiSpin, `setValue()`. So, if we change the value of roiDial, it emits a signal that connects to the roiSpin's `setValue()` function and will set the value accordingly. Here, we must note that changing either the spin or dial must change the other value because both represent a single entity. Hence, we induce a second line which calls roiDial's `setValue()` slot on changing the roiSpin's value. However, it is to be noted that the second form of connecting signals to slots is deprecated. It is given here just for reference and it is strongly discouraged to use this form. The following two lines of code execute the same for the number of years slider and spin:

```
self.yrsSlide.valueChanged.connect(self.yrsSpin.setValue)
self.connect(self.yrsSpin, SIGNAL("valueChanged(int)"), self.
yrsSlide, SLOT("setValue(int)"))
```

In order to calculate the EMI value, we connect the clicked signal of the push button to a function (slot) which calculates the EMI and displays it to the user:

```
self.connect(self.calculateButton, SIGNAL("clicked()"), self.
showEMI)
```

The EMI calculation and display function is given for your reference:

```
def showEMI(self):
  loanAmount = float(self.amtText.text())
  rateInterest = float( float (self.roiSpin.value() / 12) / 100)
  noMonths = int(self.yrsSpin.value() * 12)
  emi = (loanAmount * rateInterest) * ( ( ( (1 + rateInterest) **
noMonths ) / ( ( (1 + rateInterest) ** noMonths ) - 1) ))
  self.emiValue.display(emi)
  self.myGridLayout.addWidget(self.emiLabel, 4, 0)
  self.myGridLayout.addWidget(self.emiValue, 4, 2)
```

The sample output of the application is shown in the following screenshot:
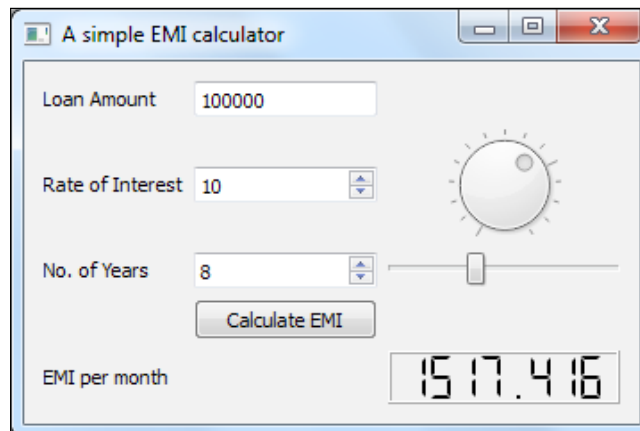
The EMI calculator application uses the predefined signals, say, for example, `valueChanged()`, `clicked()` and predefined slots, `setValue()`. However, the application also uses a user-defined slot `showEMI()` to calculate the EMI. As with slots, it is also possible to create a user-defined signal and emit it when required. The following program is an example for creating and emitting user-defined signals:

```python
import sys
from PySide.QtCore import *

# define a new slot that receives and prints a string
def printText(text):
  print(text)

class CustomSignal(QObject):
  # create a new signal
  mySignal = Signal(str)


if __name__ == '__main__':
  try:
    myObject = CustomSignal()
    # connect signal and slot
    myObject.mySignal.connect(printText)
    # emit signal
    myObject.mySignal.emit("Hello, Universe!")
  except Exception:
    print(sys.exc_info()[1])
```

This is a very simple example of using custom signals. In the `CustomSignal` class, we create a signal named `mySignal` and we emit it in the main function. Also, we define that on emission of the signal `mySignal`, the `printText()` slot would be called. Many complex signal emissions can be built this way.

Drag-and-drop

There are various ways in which you can transfer data between two objects or applications. Drag-and-drop is a modern visual technique of transformation of data between objects. It enables the user to copy and paste very intuitively. The drag-and-drop is a combination of two events, namely "Dragging and Dropping". The widgets can serve as drag sites, drop sites, or as both. One of the important factors that we should take care of is the **MIME** type of the object that we would drag-or-drop. It is to ensure that the information can be transferred safely between applications. The various MIME types supported by Qt include plain text, html text, uri-list text, image data, and color data. We will explore the Qt classes used for this action and shortly test with an example.

The various classes that are involved in drag-and-drop and their necessary MIME encoding and decoding are listed in the following table:

Class	Description
QDragEnterEvent	Provides an event which is sent to a widget when drag and drop action enters it
QDragLeaveEvent	Provides an event which is sent to a widget when drag and drop action leaves it
QDragMoveEvent	Provides an event which is sent to a widget when drag and drop action is in progress
QDropEvent	Provides an event which is sent to a widget when drag and drop action is completed
QMimeData	Provides a container for data that records information about its MIME type

A drag can be initiated by setting the widget's `setDragEnabled()` with a Boolean True value. The dropping functionality can be implemented by re-implementing the `dragMoveEvent()` and `dropEvent()`. As the user drags over the widget, `dragMoveEvent()` occur and `dropEvent()` when the drag event is completed. We will now see an example for the drag-and-drop events and the working of the code will be explained in following the code:

```
class MyWidget(QWidget):
  def __init__(self):
    QWidget.__init__(self)
    self.myListWidget1 = QListWidget()
    self.myListWidget2 = QListWidget()

    self.myListWidget2.setViewMode(QListWidget.IconMode)

    self.myListWidget1.setAcceptDrops(True)
    self.myListWidget1.setDragEnabled(True)

    self.myListWidget2.setAcceptDrops(True)
    self.myListWidget2.setDragEnabled(True)

    self.setGeometry(300, 350, 500, 150)

    self.myLayout = QHBoxLayout()
    self.myLayout.addWidget(self.myListWidget1)
    self.myLayout.addWidget(self.myListWidget2)
```

```
        l1 = QListWidgetItem(QIcon('blue_bird.png'),"Angry Bird Blue")
        l2 = QListWidgetItem(QIcon('red_bird.png'),"Angry Bird Red")
        l3 = QListWidgetItem(QIcon('green_bird.png'),"Angry Bird Green")
        l4 = QListWidgetItem(QIcon('black_bird.png'),"Angry Bird Black")
        l5 = QListWidgetItem(QIcon('white_bird.png'),"Angry Bird White")

        self.myListWidget1.insertItem(1, l1)
        self.myListWidget1.insertItem(2, l2)
        self.myListWidget1.insertItem(3, l3)
        self.myListWidget1.insertItem(4, l4)
        self.myListWidget1.insertItem(5, l5)

        QListWidgetItem(QIcon('gray_pig.png'), "Grey Pig", self.
    myListWidget2)
        QListWidgetItem(QIcon('brown_pig.png'), "Brown Pig", self.
    myListWidget2)
        QListWidgetItem(QIcon('green_pig.png'), "Green Pig", self.
    myListWidget2)

        self.setWindowTitle('Drag and Drop Example');

        self.setLayout(self.myLayout)
```

The preceding program on execution will look like the following screenshot:
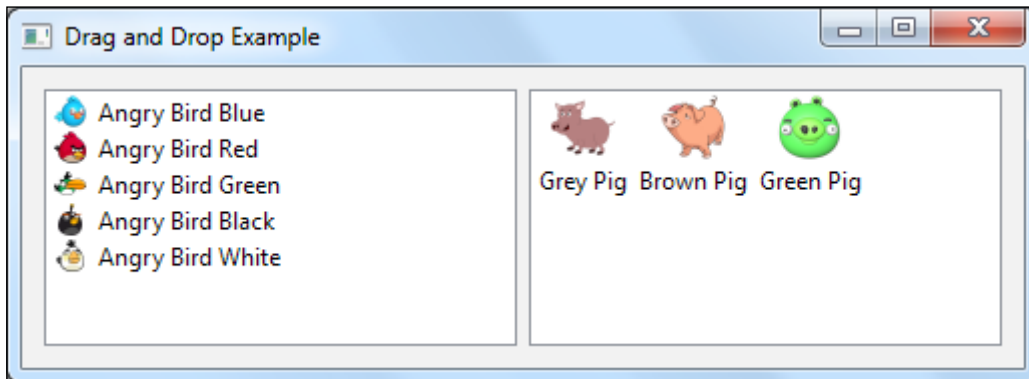
Both partition of the application has a `QListWidget` object with some items added to it. The left side is the default view mode of QListWidget and the right side is set to icon view mode. Both these widgets support dragging mode as they are set with `setDragEnabled(True)`. They also accept dropping functionality, as the `setAcceptDrops(True)` is set. You can test this by dragging-and-dropping between the widgets. We can control the behavior of the application by re-implementing the aforesaid event handler functions.

Drawing

The `PySide.QtGui.QPainter` class performs low-level painting on widgets and other paint devices. The `QPainter` class provides all the functions for drawing simple lines to more complex shapes. This class also provides settings for rendering quality images and supports clipping. The drawing is usually done within the widget's `paintEvent()` function. The drawing functionalities are placed in between the `begin()` and `end()` functions of the `QPainter` object. The `QPainter` object is initialized with the constructor, customized with some set functions, for example, pen style and brush style, and then the draw function is called. The `QPainter.isActive()` function indicates if the painter is active. The `QPainter` object is activated when `QPainter.begin()` is invoked and deactivated on calling `QPainter.end()`.

The various drawings are performed using the QPainter's draw functions.
The QPainter has three important settings that set the appearance of the drawing.
They are:

- **Pen**: The pen is used for drawing lines and shapes outlines. It takes various settings for drawing that include color, width, line style, and so on.

- **Brush**: The brush is used for pattern filling of geometric shapes. The various settings that a brush can take include color, style, texture, gradient, and so on.

- **Font**: The font is mainly used for drawing Unicode text. The font settings include font style, font family, and point size.

The settings for these parameters can be set and modified anytime by calling the `setFont()`, `setBrush()`, and `setFont()` on QPainter with their respective QPen, QBrush, or QFont objects.

In this section, we are going to explore the most commonly used drawing shapes. The following table will give you a gist of the available draw functions of the QPainter object:

Function	Description
`drawPoint()`	Draws a single point at the given position
`drawText()`	Draws the given text within the defined rectangle
`drawLine()`	Draws a line between two point pairs
`drawRect()`	Draws a rectangle by the given rectangle
`drawRoundedRect()`	Draws a rectangle with rounded edges or corners
`drawEllipse()`	Draws an ellipse defined by the given rectangle
`drawArc()`	Draws an arc defined by the given rectangle

Function	Description
`drawPie()`	Draws a pie defined by the given rectangle
`drawChord()`	Draws a chord defined by the given rectangle
`drawPolyline()`	Draws a polyline defined by the given points
`drawPolygon()`	Draws a polygon defined by the given points
`drawConvexPolygon()`	Draws a convex polygon defined by the given polygon
`drawImage()`	Draws the given Image into the given rectangle
`drawPath()`	Draws the given painter path defined by the QPainterPath
`drawPicture()`	Draws the given picture

All the earlier listed functions take various arguments as their parameters for different drawing functionalities. Also, all these drawing functions use the current pen, brush, and text settings to draw the objects. This section is not enough to cover and discuss all the different types of the drawing functions and hence we would see a sample program that is self-explanatory and exhibits the different styles of the listed functions. The complete version of the basic drawing functionality code can be downloaded from the book's site. Here, we just show the contents of the `paintEvent()` function with different drawing shapes. The complete code is bundled with event handling that we have discussed in the first section of this chapter and usage of other built-in widgets like Combo-Box that we would discuss in the next chapter. As of now, it is sufficient for you if you could understand the reimplementation of event handlers and drawing functions of the program:

```python
def paintEvent(self, event):
    rect = QRect(10, 20, 80, 60)

    startAngle = 30 * 16
    arcLength = 120 * 16

    painter = QPainter()
    painter.begin(self)
    painter.setPen(self.pen)
    painter.setBrush(self.brush)
    if self.shape == PaintArea.Line:
        painter.drawLine(rect.bottomLeft(), rect.topRight())
    elif self.shape == PaintArea.Points:
        painter.drawPoints(PaintArea.points)
    elif self.shape == PaintArea.Polyline:
        painter.drawPolyline(PaintArea.points)
    elif self.shape == PaintArea.Polygon:
```

```
        painter.drawPolygon(PaintArea.points)
    elif self.shape == PaintArea.Rect:
       painter.drawRect(rect)
    elif self.shape == PaintArea.RoundRect:
       painter.drawRoundRect(rect)
    elif self.shape == PaintArea.Ellipse:
       painter.drawEllipse(rect)
    elif self.shape == PaintArea.Arc:
       painter.drawArc(rect, startAngle, arcLength)
    elif self.shape == PaintArea.Chord:
       painter.drawChord(rect, startAngle, arcLength)
    elif self.shape == PaintArea.Pie:
       painter.drawPie(rect, startAngle, arcLength)
    elif self.shape == PaintArea.Path:
       painter.drawPath(path)
    elif self.shape == PaintArea.Text:
       painter.drawText(rect, QtCore.Qt.AlignCenter, "Basic Drawing
Widget")
    painter.end()
```

This would produce a window as given in the following screenshot. You can select from the combo boxes the choice of your drawing and it would be painted in the application:
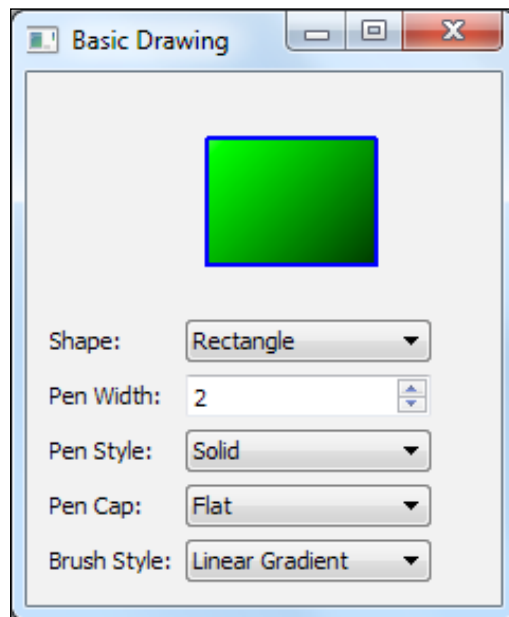
Graphics and effects

We could create any custom graphics we like by creating a custom widget and by reimplementing its paint event. This approach is very helpful when we are trying to create some small graphics like drawing graphs or for drawing basic shapes. In order to create animations and more complex graphics we will take help from the PySide's graphics view classes:

- **QGraphicsScene**: This provides a surface for managing a large number of 2D graphical items
- **QGraphicsItem**: This serves as the base class for all graphical items like ellipse, line, and so on in a graphics scene
- **QGraphicsView**: This provides a widget for displaying the contents of a graphics scene

The graphic view classes can be used by first creating a scene represented by a QGraphicsScene object. Scenes can be associated with the QGraphicsView object to represent or view on the screen. Items that are represented by the QGraphicsItem object can be added to the scene. A scene is created, items are added, and visualized in that order. QGraphicsView can be triggered to visualize a whole scene or only a part of it by changing the bounding rectangle values. The interactions can happen by using the mouse or a keyboard. The graphics view translates the mouse and key events into scene events represented by QGraphicsSceneEvent and forwarding them to the visualized scene. The custom scene interactions are achieved by re-implementing the mouse and key event handlers. The most interesting feature of the graphics views is that we can apply transformations to them, for example, scaling and rotation. This can be done without changing the original scene's items.

```python
class MyView(QGraphicsView):
  def __init__(self):
    QGraphicsView.__init__(self)

    self.myScene = QGraphicsScene(self)
    self.myItem = QGraphicsEllipseItem(-20, -10, 50, 20)
    self.myScene.addItem(self.myItem)
    self.setScene(self.myScene)

    self.timeLine = QTimeLine(1000)
    self.timeLine.setFrameRange(0, 100)
    self.animate = QGraphicsItemAnimation()
    self.animate.setItem(self.myItem)
    self.animate.setTimeLine(self.timeLine)
```

```
        self.animate.setPosAt(0, QPointF(0, -10))
        self.animate.setRotationAt(1, 360)

        self.setWindowTitle("A Simple Animation")
        self.timeLine.start()
```

This program will create an animated ellipse as its output. As discussed, we have created scene, added items to it, and showed it via the view class. The animation is supported by the `QGraphicsAnimationItem()` object. Many more complex animations can be built on top of this but explaining those is out of the scope of this book. You can explore by yourself to create more QGrapicsView objects and create complex animations.

The `PySide.QtGui.QGraphicsEffect` class serves as the base class for the graphical effects on an application. With the help of effects, we can alter the appearance of elements. The graphical effects objects operate between the sources, say, for example, a pix map item and the destination, the viewport, and render the respective effects for the image. The various graphical effects that Qt provides are:

- **QGrpahicsBlurEffect**: It blurs the item by a given radius
- **QGraphicsDropShadowEffect**: It renders a drop shadow behind the item
- **QGraphicsColorizeEffect**: It renders the item in shades of any given color
- **QGraphicsOpacityEffect**: It renders the item with an opacity

All these classes are inherited from the base class `QGraphicsEffect`. The following code excerpt shows an example of adding effects to the items:

```
        self.effect = QGraphicsDropShadowEffect(self)
        self.effect.setBlurRadius(8)
        self.myItem.setGraphicsEffect(self.effect)
        self.myItem.setZValue(1)
```

When this effect is added, you will notice a shadow in the animated ellipse as shown in the following screenshot. Similarly, other effects can be added to the items.
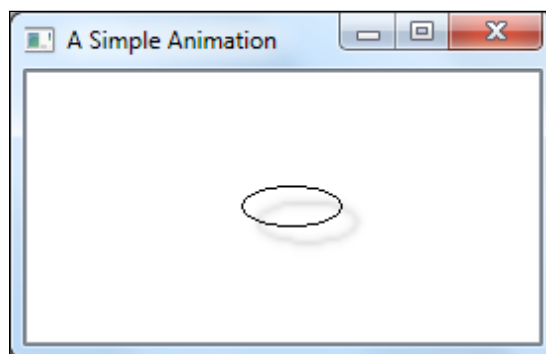
Summary

This chapter would have been a real roller-coaster ride as we were taken into some internal depth of the PySide programming. We started with discussing event handlers and reimplementation techniques to achieve the task at hand. We also discussed about event filters and re-implementing the `notify()` function. Unless absolutely necessary the latter forms of re-implementing events should be avoided to make an efficient program.

We then explored the very fundamental mechanism to Qt, signals, and slots. The signals and slots mechanism follow an observer pattern listening and binding to objects when called. We started with implementing the built-in signals and slots. Later in this section, we implemented and emitted our own custom signals and also discussed how to listen to them.

In the latter half, we shifted our focus to diagrams and graphics. Starting with the drag-and-drop functionality usage, we have also seen various types of QPainter draw objects. The chapter ended with a brief discussion on graphics and effects. The examples that are shown in the latter half of the chapter are very basic examples to help you understand the basic concepts. Much more complex applications can be designed and it would by itself be a subject matter for a complete book.

5
Dialogs and Widgets

Any main windowed applications in the GUI are to be supplemented with dialogs and widgets to present a complete workable and usable application to the users. Dialogs are those small sized windows with some specific functions that will aid the users with selecting some options or executing some operations. The most common examples of dialogs include "File Open" dialog and a "Search Dialog" in many text/image editing applications, a "Color Chooser" dialog in various paint applications and so on. In many GUI toolkits, the terms dialogs and widgets are used interchangeably. As a general difference, the dialogs are the small windows whose main purpose is to establish a connection between the user and the program. We normally use dialog boxes to receive an input from the users or to represent an output or error message to the users. However, the widgets are collections of building blocks of the applications such as buttons, checkboxes, progress bars, and so on. This chapter will introduce you to some of the built-in dialogs and widgets that Qt provides us. Moving on, we will develop a customized "Find" dialog that will add to the text editor program which we have developed in third chapter. We will also develop a customized "Analog Clock" widget.

Built-in dialogs

`Qt` is provided with a rich set of built-in dialogs. They are as follows:

- `QFileDialog`: It provides the user with a dialog that allows them to select files or directories
- `QErrorMessage`: It provides the user with a dialog that displays error messages
- `QColorDialog`: It provides the user with a dialog for specifying/choosing between colors
- `QPrintDialog`: It provides the user with a dialog that aids in printer and printing configuration

- `QPageSetupDialog`: It provides the user with a dialog that manages configuration for the page-related options on a printer
- `QWizard`: It provides a dialog framework for wizards
- `QProgressDialog`: It provides feedback on the progress of a slow operation
- `QPrintPreviewDialog`: It provides a dialog for previewing and configuring page layouts for printer output
- `QMessageBox`: It provides a dialog for displaying some information to the user
- `QInputDialog`: It provides a simple convenience dialog to get a single value from the user
- `QFontDialog`: It provides a dialog widget for selecting a font

In this section, we would discuss the implementations of some widely used widgets. A few of the widgets like, `QMessageBox`, `QFontDialog`, and `QErrorMessage` have already been introduced to you in some of the preceding chapters.

QFileDialog

The `PySide.QtGui.QFileDialog` class provides a dialog that allows users to select files or directories. It helps users to traverse the native file system in order to select one or many files or directory. The easy and best way to create a file dialog is to use the static functions provided by the `QFileDialog` class. A sample use of static function is as follows:

```
fileName = QFileDialog.getOpenFileName(self, "Open Files", "c:/",
  "Textfiles(*.txt)")
```

In this example, the file dialog was created using a static function `getOpenFileName`. Initially, this function call will create a file dialog with the path mentioned in the third parameter of the function. The fourth parameter restricts the type of files that has to be shown in the dialog for opening. The first parameter takes the value of the parent to the dialog and the second is used to display a title for the dialog. The filters that are represented in the fourth parameter can take multiple values depending on the type of files that are to be filtered. Please note that the values must be separated by a delimiter ";". An example for the filter can look like as shown in the following code line.

```
"Images (*.png *.jpg);; Text files (*.txt);; Word
  documents(*.doc)"
```

The following code shows an example for creating the "File" dialog as a menu option. The program is self-explanatory and is based on the concepts that we have seen in the third chapter of this book.

```python
import sys
from PySide.QtGui import *


class MyFileDialog(QMainWindow):

  def __init__(self):
    QMainWindow.__init__(self)

    self.textEdit = QTextEdit()
    self.setCentralWidget(self.textEdit)
    self.statusBar()

    openFile = QAction(QIcon('open.png'), 'Open', self)
    openFile.setShortcut('Ctrl+O')
    openFile.setStatusTip('Open new File')
    openFile.triggered.connect(self.showDialog)

    menubar = self.menuBar()
    fileMenu = menubar.addMenu('&File')
    fileMenu.addAction(openFile)

    self.setGeometry(300, 300, 350, 300)
    self.setWindowTitle('Example - File Dialog')
    self.show()

  def showDialog(self):
    fileName, _ = QFileDialog.getOpenFileName(self, "Open Text
      Files", "c:/", "Text files(*.txt)")

    contents = open(fileName, 'r')

    with contents:
      data = contents.read()
      self.textEdit.setText(data)
```

```
if __name__ =='__main__':
  # Exception Handling
  try:
    myApp = QApplication(sys.argv)
    myFD = MyFileDialog()
    myFD.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

If you run this program, you could witness a file open dialog is opened on triggering the **File | Open** option in the menu bar.

The file dialog can also be created without using the static functions by directly creating an instance of the QFileDialog class as explained in the following code snippet:

```
fileDialog = QFileDialog(self)
fileDialog.setFileMode(QFileDialog.AnyFile)
fileDialog.setNameFilter("Text files(*.txt)")
```

The second line of the preceding code sets the mode of the file dialog to AnyFile, which means that the user can specify a file that doesn't even exist in the file system. This mode is highly useful when we want to create a "Save As" dialog. The other modes that the file dialog can take are:

- QFileDialog.ExistingFile: If the user must select an existing file
- QFileDialog.Directory: If the user must select only a directory and not files
- QFileDialog.ExistingFiles: If the user wants to select more than one file

The third line depicts how to set the filters for the file dialog as explained in the previous program. Also, the file dialog has two view modes, namely, View and Detail, where the latter being the default mode. As the name indicates, the list view just displays the name of the files and directories in a list but the detail mode enhances it with additional details about the file such as, file size, date modified, and so on.

QInputDialog

The `PySide.QtGui.QInputDialog` class provide a very easy and convenient dialog to receive input from the users. The input can be a text string, a number or an item from the list. A label is provided with the input box to indicate the user what they have to enter. To enable this, four convenient functions are used:

- `QInputDialog.getText()`: It receives a text or string from the user
- `QInputDialog.getInteger()`: It receives an integer value as an input
- `QInputDialog.getDouble()`: It receives a float value as input with double precision accuracy
- `QInputDialog.getItem()`: It receives a particular selectable value from the list of items

The dialog is provided with two buttons **OK** and **Cancel** to accept or reject values respectively as shown in the following screenshot:
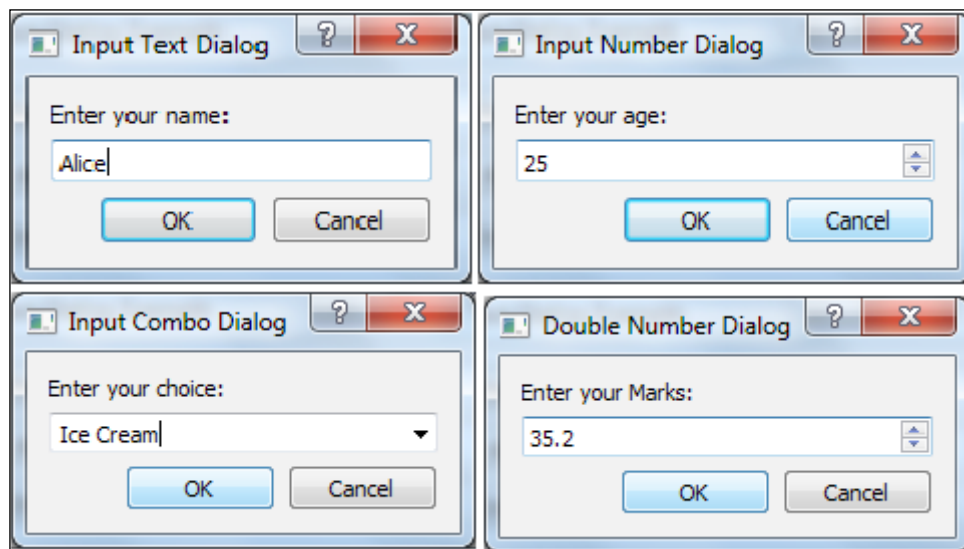
The following code explains the use of various input dialogs:

```
# Import necessary modules
import sys
from PySide.QtGui import *


class MyInputDialog(QWidget):
```

```python
def __init__(self):

    QWidget.__init__(self)
    self.myNameButton = QPushButton('Name', self)
    self.myNameButton.clicked.connect(self.showNameDialog)

    self.myAgeButton = QPushButton('Age', self)
    self.myAgeButton.clicked.connect(self.showAgeDialog)

    self.myChoiceButton = QPushButton('Choice', self)
    self.myChoiceButton.clicked.connect(self.showChoiceDialog)

    self.myNameLE = QLineEdit(self)
    self.myAgeLE = QLineEdit(self)
    self.myChoiceLE = QLineEdit(self)

    self.myLayout = QFormLayout()
    self.myLayout.addRow(self.myNameButton, self.myNameLE)
    self.myLayout.addRow(self.myAgeButton, self.myAgeLE)
    self.myLayout.addRow(self.myChoiceButton, self.myChoiceLE)

    self.setLayout(self.myLayout)
    self.setGeometry(300, 300, 290, 150)
    self.setWindowTitle('Input Dialog Example')
    self.show()

def showNameDialog(self):
    text, ok = QInputDialog.getText(self, 'Input Text Dialog',
        'Enter your name:')

    if ok:
        self.myNameLE.setText(str(text))

def showAgeDialog(self):
    text, ok = QInputDialog.getInteger(self, 'Input Number Dialog',
        'Enter your age:')

    if ok:
        self.myAgeLE.setText(str(text))

def showChoiceDialog(self):
    strList = ['Ice Cream', 'Chocolates', 'Milk Shakes']
    text, ok = QInputDialog.getItem(self, 'Input Combo
        Dialog',
        'Enter your choice:', strList)

    if ok:
        self.myChoiceLE.setText(str(text))
```

```
if __name__ =='__main__':
  # Exception Handling
  try:
    myApp = QApplication(sys.argv)
    myID = MyInputDialog()
    myID.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

In the preceding code, each button click event is connected to a slot that presents the user with different input dialogs. The values of the dialogs reflect on the line edit box after clicking on the **OK** button. A sample output of the preceding program is given in the following screenshot for your reference:
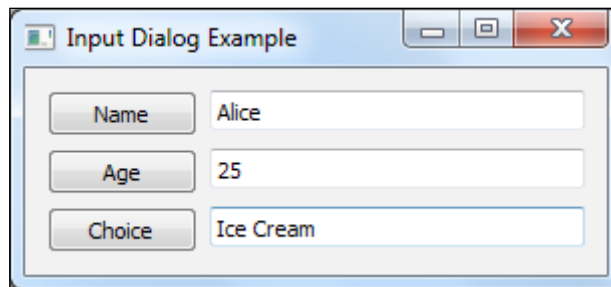
QColorDialog

The `PySide.QtGui.QColorDialog` class provides a dialog for choosing and specifying colors. The color dialog is mainly used in the paint applications to allow the user to set the color of the brush or paint an area with a specific color. This dialog can also be used to set text colors in text based applications. As with the other dialogs, we use a static function `QColorDialog.getColor()` to show the dialog and subsequently allow the user to select a color from the dialog. This dialog can also be used to allow the users to select the color transparency by passing some additional parameters. It is also possible to set and remember the custom colors and share them between color dialogs in an application. The following code is a short example of using the color dialog:

```
# Import necessary modules
import sys
from PySide.QtGui import *
```

```python
class MyColorDialog(QWidget):

  def __init__(self):
    QWidget.__init__(self)
    myColor = QColor(0, 0, 0)

    self.myButton = QPushButton('Press to Change Color', self)
    self.myButton.move(10, 50)

    self.myButton.clicked.connect(self.showColorDialog)

    self.myFrame = QFrame(self)
    self.myFrame.setStyleSheet("QWidget { background-color: %s }"
      % myColor.name())
    self.myFrame.setGeometry(130, 22, 100, 100)

    self.setGeometry(300, 300, 250, 180)
    self.setWindowTitle('Color Dialog - Example')
    self.show()

  def showColorDialog(self):

    color = QColorDialog.getColor()

    if color.isValid():
      self.myFrame.setStyleSheet("QWidget { background-color: %s
        }"
        % color.name())

if __name__ =='__main__':
  # Exception Handling
  try:
    myApp = QApplication(sys.argv)
    myCD = MyColorDialog()
    myCD.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

The preceding example will show a button, which when clicked shows a color dialog. The colour can be selected and the same is painted in the frame used for this purpose.

QPrintDialog

The `PySide.QtGui.QPrintDialog` class provides a dialog for specifying the user's printer configuration. The configuration include document-related settings, such as the paper-size and orientation, type of print, color or grayscale, range of pages, and number of copies to print. The configuration settings also allows the user to select the type of printer from the printers available, including any configured network printers. The `QPrintDialog` objects are constructed with a `PySide.QtGui.QPrinter` object and executed using the `exec_()` function. The printer dialog uses the native system of display and configuration. The following example shows a crude way of creating a print dialog. The refined implementation is left as an exercise for you to explore. A sample implementation can be downloaded from the code snippets that come along with this book.

```
def printDoc(self):
    document = QTextDocument("Sample Page")
    printer = QPrinter()

    myPrintDialog = QPrintDialog(printer, self)
    if myPrintDialog.exec_() != QDialog.Accepted:
      return
      document.print_(printer)
```

Custom dialogs

We have seen some examples of the built-in dialogs in the previous section. The need may arise in a real application scenario to define and create a custom dialog based on the user's requirement. `Qt` provides the support for creating custom based dialogs and use it in addition to the various built-in dialogs. In this section, we are going to explore how to create a custom "Find" dialog for our text editor application that we have created in *Chapter 3*, *Main Windows and Layout Management*. The `FindDialog` class is inherited from the `QDialog` class and defines the properties for implementing a search function. Once the find dialog functions are defined, it can be added to our text editor application and the slots are implemented accordingly.

In order to create a find dialog, we must create an outline of how it looks. The following screenshot is a sample look of how we would want our find dialog to appear:
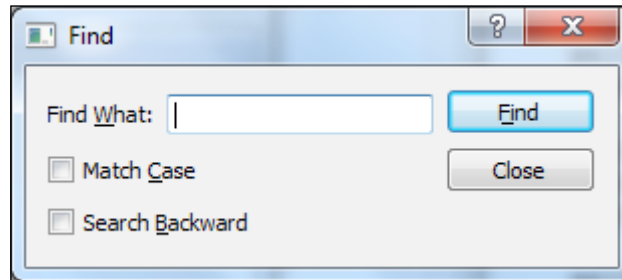
This is a very simple find dialog. We would want to capture the text to be searched by using a line edit dialog. The two checkboxes, **Match Case** and **Search Backward** try to catch the user's choices. The button click events are connected to specific slots to perform the desired action. We have used the layout concepts introduced to you in the previous chapter to place the widgets.

```python
class FindDialog(QDialog):

  def __init__(self):
    QDialog.__init__(self)
    self.findLabel = QLabel("Find &What:")
    self.lineEdit = QLineEdit()
    self.findLabel.setBuddy(self.lineEdit)

    self.caseCheckBox = QCheckBox("Match &Case")
    self.backwardCheckBox = QCheckBox("Search &Backward")

    self.findButton = QPushButton("&Find")
    self.findButton.setDefault(True)
    self.closeButton = QPushButton("Close")

    self.topLeftLayout = QHBoxLayout()
    self.topLeftLayout.addWidget(self.findLabel)
    self.topLeftLayout.addWidget(self.lineEdit)

    self.leftLayout = QVBoxLayout()
    self.leftLayout.addLayout(self.topLeftLayout)
    self.leftLayout.addWidget(self.caseCheckBox)
    self.leftLayout.addWidget(self.backwardCheckBox)
```

```
        self.rightLayout = QVBoxLayout()
        self.rightLayout.addWidget(self.findButton)
        self.rightLayout.addWidget(self.closeButton)
        self.rightLayout.addStretch()

        self.mainLayout = QHBoxLayout()
        self.mainLayout.addLayout(self.leftLayout)
        self.mainLayout.addLayout(self.rightLayout)

        self.findButton.clicked.connect(self.findText)
        self.setWindowTitle("Find")
        self.setLayout(self.mainLayout)
        self.show()

    def findText(self):
        mySearchText = self.lineEdit.text()
        if self.caseCheckBox.isChecked():
            caseSensitivity = Qt.CaseSensitive
        else:
            caseSensitivity = Qt.CaseInsensitive
        if self.backwardCheckBox.isChecked():
        #search functionality goes here...
            print("Backward Find ")
        else:
        #search functionality goes here...
            print("Forward Find")
In order to use this dialog, simply create an instance of
    MyFindDialog that will create a 'find' dialog as shown in the
    below lines of code.
def findDialog(self):
        myFindDialog = FindDialog()
        myFindDialog.exec_()
```

Thus, we can create and customize dialog according to the needs of our application.

Widgets at a glance

Widgets are the basic building blocks of a Graphical User Interface application, which are combined or grouped in a way that helps in interaction and manipulation. `Qt` comes with a variety of basic and advanced built-in widgets that can be customized to our own needs. A list of predefined widgets is given in the following tables for your reference. `QWidget` is the base class for all the widgets given in the list.

Basic widgets

The basic widgets section contains the list of widgets that are simple and designed for direct use. The following list contains the available basic widgets, most of which we have used in the examples that we have seen so far:

Widget	Description
QCheckBox	Provides an option button that can be switched on or off. Associated with a text label.
QComboBox	Provides a pop-up list and a combined button by optimizing the space used for providing a list of options.
QCommandLinkButton	Provides a Vista style command link button similar to that of a radio button used to choose between a set of mutually exclusive options.
QDateEdit	Provides a widget for editing dates. It is inherited from QDateTime.
QDateTimeEdit	Provides a widget for editing dates and times by using the keyboard.
QDial	Provides a rounded range control widget similar to the ones used in speedometer or potentiometer. Several notches differentiate the main and sub values.
QDoubleSpinBox	Provides a spin box widget that takes doubles. Allows to choose a value by clicking up and down buttons or arrow keys
QFocusFrame	Provides a focus frame which can be outside of a widget's normal paintable area.
QFontComboBox	Provides a combo box widget whose contents are different font family styles in a alphabetized list.
QLCDNumber	Provides a widget that displays the number in LCD like form.
QLabel	Provides an image or text display whose visual appearance can be configured in various ways. It supports plain text, rich text, pixmap, movie, or a number.
QLineEdit	Provides a one-line text editor used to edit a single line of text.
QMenu	Provides a menu widget(selection menu) for use in menu bars, context menus, and other pop-up menus.
QProgressBar	Provides a progress bar widget that helps to indicate the users with the completion status of an activity
QPushButton	Provides a button that is usually executes a command on push or click.

Widget	Description
QCheckBox	Provides an option button that can be switched on or off. Associated with a text label.
QRadioButton	Provides a radio button option that can be switched on or off associated with a text label.
QScrollArea	Provides a scrolling view that is used to display the contents of a child widget within a frame and also provides scroll bars if the widget exceeds the size of the frame.
QScrollBar	Provides a vertical or a horizontal scroll bar that enables the user to access parts of a document that is larger than the widget used to display it.
QSizeGrip	Provides a standard windows resize handle for resizing top-level windows.
QSlider	Provides a vertical or horizontal slider used for controlling a bounded value. It lets the user move a slider along a horizontal or vertical groove.
QSpinBox	Provides a spin box widget designed to handle integers and discrete set of values.
QTabBar	Provides a tab bar to be used in tabbed dialogs.
QTabWidget	Provides a tab bar and a page area that is used to display pages related to each tab in a stack of tabbed widgets.
QTimeEdit	Provides a widget for editing times inherited by the QDateTimeEdit widget.
QToolBox	Provides a widget that displays a column of tabs one above the other with the current item displayed below the current tab.
QToolButton	Provides a special quick-access button to specific commands or options associated with an icon.

Advanced widgets

Advanced widgets provide more complex user interface controls, which may be used in creating various advanced applications. These widgets make the work of programmers easy by providing us with the most common application features as widget libraries.

Widget	Description
QCalendarWidget	Provides a monthly based calendar widget for date selection initialized with the current month and year.
QColumnView	Provides a widget for model/view implementation of a column view, also referred as cascading list.
QDataWidgetMapper	Provides a widget for mapping between a section of a data model to widgets which is used to create data-aware widgets.
QDesktopWidget	Provides a widget for accessing screen information on multi-head systems; used for managing the physical screen space in user's desktop.
QListView	Provides a widget to present items stored in a model, either as a simple non-hierarchical list or a collection of items.
QTableView	Provides a widget to implement a table view that displays items from a model that represents standard tables.
QTreeView	Provides a widget to implement a tree representation of items from a model that provides standard hierarchical lists.
QUndoView	Provides a widget which displays the list of commands pushed on to a undo stack that has the most recently executed commands in order.
QWebView	Provides a widget that is used to view and edit web documents.

Organizer widgets

The following widgets are mainly used for organizing and grouping various basic and primitive widgets into more complex applications and dialogs:

Widget	Description
QButtonGroup	Provides an abstract container into which button widgets can be placed.
QGroupBox	Provides a group box frame with a title that acts as a place holder for various other widgets.
QSplitter	Provides implementation for a splitter that lets the user control the size of child widgets by dragging the boundary between the child widgets.
QSplitterHandle	Provides a handle functionality for the splitter to resize the windows.
QStackedWidget	Provides a stack of widgets where only one widget is available at a time similar to a tab widget.
QTabWidget	Provides a stack of tabbed widgets.

We have seen the implementation of few of the built-in widgets given in the preceding table in our previous chapters. One of the greatest strengths of PySide lies in the ease of creation of customized widgets. We can group together some of the basic widgets to create a customized widget on our own. Before we could do that, we also have several ways to customize a widget to suit our needs. The basic form of customization is to change the properties of the existing widget. We can also opt to use stylesheets to customize the widget's appearance and some aspects of its behavior. In some cases, it is highly likely that we may require a widget that is different from any of the standard widgets. In those cases, we can subclass `QWidget` directly and can completely define the behavior and appearance of the widget ourselves. In the example that follows this text, we create a customized "Analog Clock" widget and demonstrate how to create custom widgets.

Custom widgets

In this section we see an implementation of a customized widget.

To start with we define the `AnalogClock` class inherited from `QWidget`. We define two variables that would be used for drawing the `hourHand` and `minuteHand` of the analog clock. Also, we define the colors for the pointers.

```
class AnalogClock(QWidget):
  hourHand = QPolygon([
    QPoint(7, 8),
    QPoint(-7, 8),
    QPoint(0, -40)
  ])

  minuteHand = QPolygon([
    QPoint(7, 8),
    QPoint(-7, 8),
    QPoint(0, -70)
  ])

  hourColor = QColor(255, 0, 127)
  minuteColor = QColor(0, 127, 127, 255)
```
Next, we define an init function that will start the timer that would update the clock on expiry of every minute. It also resizes the widget and sets a title for it.
```
def __init__(self, parent=None):
    QWidget.__init__(self)

    timer = QTimer(self)
    timer.timeout.connect(self.update)
    timer.start(1000)

    self.setWindowTitle("Analog Clock")
    self.resize(200, 200)
```

The core functionality of the analog clock is defined in the `paintEvent()` function, which would be called on update function of the analog clock widget. We call the `QTime.CurrentTime()` function to update the current time from the system. The next set of lines will set the pen properties and draws the minute hand and hour hand polygons, along with the line indications of the analog clock.

```
def paintEvent(self, event):
    side = min(self.width(), self.height())
    time = QTime.currentTime()

    painter = QPainter(self)
    painter.setRenderHint(QPainter.Antialiasing)
```

```
    painter.translate(self.width() / 2, self.height() / 2)
    painter.scale(side / 200.0, side / 200.0)

    painter.setPen(Qt.NoPen)
    painter.setBrush(AnalogClock.hourColor)

    painter.save()
    painter.rotate(30.0 * ((time.hour() + time.minute() /
60.0)))
    painter.drawConvexPolygon(AnalogClock.hourHand)
    painter.restore()

    painter.setPen(AnalogClock.hourColor)

    for i in range(12):
      painter.drawLine(88, 0, 96, 0)
      painter.rotate(30.0)

    painter.setPen(Qt.NoPen)
    painter.setBrush(AnalogClock.minuteColor)

    painter.save()
    painter.rotate(6.0 * (time.minute() + time.second() /
60.0))
    painter.drawConvexPolygon(AnalogClock.minuteHand)
    painter.restore()

    painter.setPen(AnalogClock.minuteColor)

    for j in range(60):
      if (j % 5) != 0:
        painter.drawLine(92, 0, 96, 0)
      painter.rotate(6.0)
```

On running the preceding application, we would see an analog clock widget drawn as given in the following screenshot:
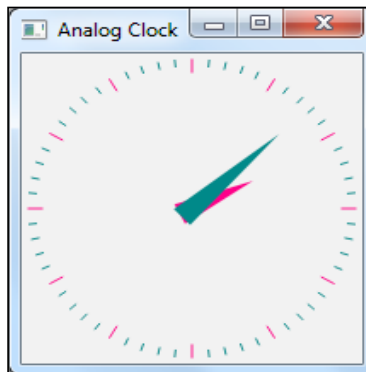
Implementation of MDI

We have already discussed about the differences between SDI and MDI applications in *Chapter 3*, *Main Windows and Layout Management*. We have seen many implementations of SDI applications. In this section, we will explore a technique of creating MDI applications.

A Multiple Document Interface application will be a main windowed application with their central widgets can be one of `PySide.QtGui.QMdiArea` or `PySide.QtGui. QWorkSpace` widget. They are by itself a widget component, which manages the central area of main window to arrange the MDI windows in a layout. Sub-windows can be created and added to the MDI area or a workspace. An example of the MDI application is as follows:

```python
class MyMDIApp(QMainWindow):

  def __init__(self):
    QMainWindow.__init__(self)

    workspace = QWorkspace()
    workspace.setWindowTitle("Simple WorkSpace Example")

    for i in range(5):
      textEdit = QTextEdit()
      textEdit.setPlainText("Dummy Text " * 100)
      textEdit.setWindowTitle("Document %i" % i)
      workspace.addWindow(textEdit)

    workspace.tile()
    self.setCentralWidget(workspace)

    self.setGeometry(300, 300, 600, 350)
    self.show()
```

The MDI windows inside a main windowed application can be set in a cascade or tile layout by default or a customized layout can be specified. The following screenshots show the two types of layout of the example application:
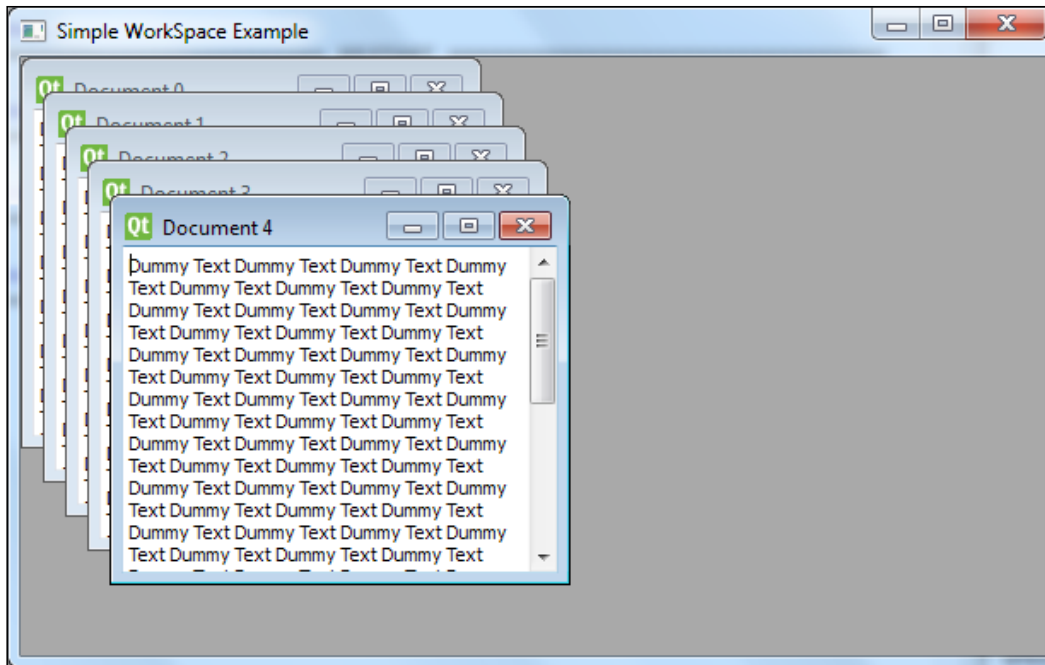
The preceding screenshot shows the placement of windows in a cascaded view. The following one implements the same example but in a tile view:
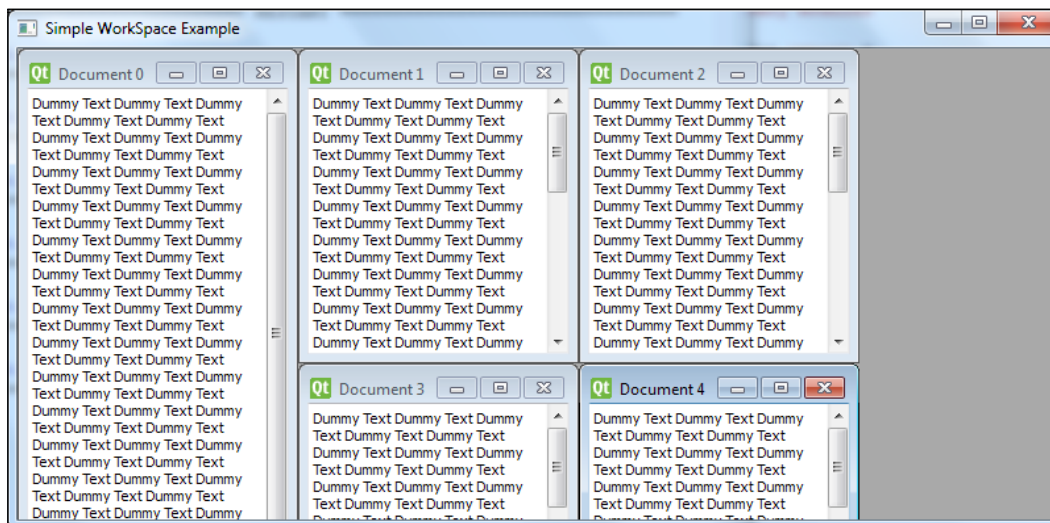
Summary

The beauty of any programming language lies in its capability to customize things for better use other than its rich library of predefined objects. PySide exhibits its dynamics by allowing the users to explore the variety of predefined dialogs and widgets and also makes it easier to create and use customized ones. This chapter has taken you through the various dialogs and widgets and also taught you how to build them on your own.

6
Handling Databases

Most of the applications that we use in our day-to-day life have to deal with some type of stored data that can be used as and when required. Databases help us to organize this stored data and also provide us some guidelines on how the data should be structured and presented to the end users. In short, databases are the organized body of related information. The software controlling the access to databases and which enables you to create, edit, and maintain them are known as the **Database Management Systems** (**DBMS**). There are numerous DBMS systems available in the market and each of them are used to suit some specific needs.

The DBMS that we use in this chapter for providing examples is **SQLite**. As the name specifies SQLite is a light-weight, public-domain, in-process database system that is used for local storage in many software applications. This chapter assumes the familiarity of users with SQL and some knowledge in model/view programming. The **QtSql** module of PySide provides a database-independent interface for accessing SQL databases. So, apart from the initial connection and some other raw SQL syntax, almost all other queries work with any SQL database.

Connecting to the database

A database connection is represented by the `PySide.QtSql.QSqlDatabase` class. This class provides an interface for accessing a database through the connection. The connection is accomplished through Qt's supported database drivers which are derived from the `PySide.QtSql.QSqlDriver` class. The `QSqlDriver` class is an abstract base class for accessing specific SQL databases, and should be used with `QSqlDatabase`. As with any other PySide feature that supports customization, it is also possible to create our own SQL drivers by sub-classing the `QSqlDriver` class and re-implementing its virtual functions.

Before explaining the concept of connecting to databases, we will see the types of database access provided by PySide. The access is provided at two levels, a high-level access using `QSqlTableModel` or `QSqlRelationalTableModel`, and a low-level access using `QSqlQuery`. The former one uses model/view architecture and provides a high-level interface for reading and writing database records from a table. The latter one is based on executing and manipulating SQL queries directly on the `QSqlDatabase` class. This class can accept DML statements, such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, as well as DDL statements, such as `CREATE`, `ALTER`, and `DROP`. It can also be used to execute database-specific commands which are not standard SQL, for example, `SET DATESTYLE=ISO` for `PostgreSQL`.

The connection to the database in an application can take at any point when it is required to. But, in most of the applications it is done after the creation of the `QApplication` object, and before the main form is created or shown. This way, we can inform users ahead of the unsuccessful connection and thus prevent them from executing database specific actions that will otherwise go waste. A connection is created by calling the static `QSqlDatabase.addDatabase()` function, by giving the name of the database driver that we want to use and optionally, a connection name. A database connection is identified by the connection name that we specify and not by the database name. It is possible for us to create multiple database connections at a time and differentiate them by the connection name. It is also correct to create a connection without a name, by making it a default connection. Once a connection object has been created, we can move on to set other necessary attributes such as, database's name, username, password, host, and port details, and other connect options, if necessary. Once this is complete, we can activate the connection to the database with the help of the `QSqlDatabase.open()` function call. The following code presents the series of steps that we have discussed in creating and activating a connection:

```
db = QSqlDatabase.addDatabase("QMYSQL")
db.setHostName("testServer")
db.setDatabaseName("sampledb")
db.setUserName("test")
db.setPassword("pass123")
ok = db.open()
```

In the preceding code, the `addDatabase()` function can be given an optional second parameter which specifies the connection name given as follows:

```
db = QSqlDatabase.addDatabase("QMYSQL", "myConn")
```

However, when using SQLite, it is enough to specify only the database name. Usually, this will be a filename but it can be the special name ":memory:" to specify using an in-memory database. When the `open()` call is executed on the connection, the file will be created if it does not exist. So, the code for connecting to the SQLite database using the SQLite driver for PySide, QSQLITE is:

```
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName("test.db")
ok = db.open()
```

This will create a connection with the SQLite database file `test.db`. We should take care of handling the connection errors by checking the return value of the `db.open()` function. It is good practice to notify the users upfront about any errors.

Once the connection is established, we can get the list of the tables, its primary index, and meta-information about the table's fields with their specific functions. The `QSqlDatabase.removeDatabase()` function is called with a connection name to remove a connection.

Executing SQL queries

After the successful connection to the database, we can execute SQL queries to perform some actions on it. If we don't specify a connection name, the default connection is taken. The `PySide.QtSql.QSqlQuery` class provides a means of executing and manipulating SQL databases.

Executing a query

The SQL query can be executed by creating an `QSqlQuery` object and calling an `exec_()` function on that. As an example, we create a table named `employee` and define its columns as follows:

```
myQuery = QSqlQuery()

myQuery.exec_("""CREATE TABLE employee (id INTEGER PRIMARY KEY
AUTOINCREMENT UNIQUE NOT NULL, first_name CHAR(20) NOT NULL, last_name
CHAR(20), age INT, sex CHAR(1), income FLOAT)""")
```

This will create a table with six fields namely, `id`, `first_name`, `last_name`, `age`, `sex`, and `income`. The `QSqlQuery` constructor accepts an optional parameter, a `QSqlDatabase` object that specifies which database connection to use. Since we don't specify any connection name in the preceding code, the default connection is used. In case of any errors, the `exec_()` function returns false and the error details are available in `QSqlQuery.lastError()`.

Inserting, updating, and deleting records

In this section we shall look at the different ways we can perform the DML commands. A simple form of inserting the values in the table that we have created in our previous section is given as follows:

```
myQuery = QSqlQuery()
myQuery.exec_("""INSERT INTO employee (first_name, last_name, age,
sex, income) VALUES ('Alice', 'M', 30, 'F', 5000.00) """)
```

This would insert a single row in the employee table. This method is easier if we need to insert a single row into the table. However, if we are required to create many rows, it is advisable to separate the query from the actual values being used. This can be achieved with the use of placeholders by binding the values with the columns in the table. Qt supports two types of placeholder systems, named binding and positional binding. This way of constructing queries is also called prepared queries.

An example of the named binding is given as follows:

```
myQuery.prepare("INSERT INTO employee (first_name, last_name, age,
sex, income) VALUES (:first_name, :last_name, :age, :sex, :income)")

for fname, lname, age, sex, income in data:
  myQuery.bindValue(":first_name", fname)
  myQuery.bindValue(":last_name", lname)
  myQuery.bindValue(":age", age)
  myQuery.bindValue(":sex", sex)
  myQuery.bindValue(":income", income)

myQuery.exec_()
```

You may note that the id column is omitted during the rows in the previous examples since we have defined it to AUTOINCREMENT values while creating the table. Now, let's look at the other type of prepared query, positional binding:

```
myQuery.prepare("INSERT INTO employee (first_name, last_name, age,
sex, income) VALUES (?, ?, ?, ?, ?)")

for fname, lname, age, sex, income in data:
  myQuery.addBindValue(fname)
  myQuery.addBindValue (lname)
  myQuery.addBindValue (age)
  myQuery.addBindValue (sex)
  myQuery.addBindValue (income)

myQuery.exec_()
```

Both the methods works with all database drivers provided by Qt. Prepared queries improve performance on databases that support them. Otherwise, Qt simulates the placeholder syntax by preprocessing the query. The actual query that gets executed can be received by calling the `QSqlQuery.executedQuery()` function. Also, note that, you need to call `QSqlQuery.prepare()` only once and you can call `bindValue()` or `addBindValue()` followed by the `exec_()` as many times as necessary. Another advantage of the prepared queries besides performance is that, we can specify arbitrary values without worrying about loosing the special characters.

`QSqlQuery` can execute any arbitrary SQL statements like `SELECT` and `INSERT` statements. So, updating and deleting records is as easy as executing the corresponding queries. For example, we can update a record as shown in the following line:

```
myQuery.exec_("UPDATE employee SET income=7500.00 WHERE id=5")
```

Similarly, we can delete a record by:

```
myQuery.exec_("DELETE FROM employee WHERE id=8")
```

Successfully executed SQL statements set the query's state to active and can be retrieved from `QSqlQuery.isActive()`. Otherwise, it is set to inactive. This method will return a Boolean value, True or False, depending on the success of the operation.

Navigating records

The next feature that we are about to discuss is how to navigate the records of the result set of a `SELECT` query. Navigating the records is performed by the following functions:

- `PySide.QtSql.QSqlQuery.next()`
- `PySide.QtSql.QSqlQuery.previous()`
- `PySide.QtSql.QSqlQuery.first()`
- `PySide.QtSql.QSqlQuery.last()`
- `PySide.QtSql.QSqlQuery.seek()`

These functions help us in iterating back and forth the records. However, if we need to move only forward through the results, we can set `QSqlQuery.setForwardOnly()` which can improve performance, and saves significant amount of memory in some databases. The `QSqlQuery.value()` functions takes an integer positional argument, which returns the value of field index in the current record. The fields are numbered from left to right using the text of the `SELECT` statement. For example, in the following query, field 0 represents the `first_name` and field 1 represents the `last_name`:

```
SELECT first_name. last_name FROM employee
```

Since `QSqlQuery.value()` takes an index positional argument, it is not advised to use `SELECT *` in the query; instead use the column names because we don't know the order of columns in the `SELECT *` query.

Let us now look at an example of navigating the records through the result set:

```
myQuery.exec_("SELECT id, first_name, income FROM employee")
while myQuery.next():
id = myQuery.value(0).toInt()
name = myQuery.vaue(1).toString()
salary = myQuery.value(2).toInt()
```

In the preceding example, you would have noted that we use the `toInt()` and `toString()` functions to convert the result to specific data type, since all the values that are returned are of QVariant type which can hold various data types such as int, string, datetime, and so on.

Before closing the section on executing SQL queries, we will have a look at few more useful functions that the `QSqlQuery` class offers.

The `QSqlQuery.numRowsAffected()` function will return the number of rows that are affected by the result of an `UPDATE` or `DELETE` query. This function returns `-1` if it cannot be determined or the query is not active. In case of `SELECT` statements, this function returns undefined, instead we use `QSqlQuery.size()` that will return the size of the result set. This function also returns `-1` if the size cannot be determined or if the database does not support reporting information about query sizes or if the query is not active.

`QSqlQuery.finish()` will instruct the database driver that no more data will be fetched from the query until it is re-executed. Usually, we do not call this function until we want to free some resources such as locks or cursors if you intend to reuse the query at a later time. Finally, we can call `QSqlQuery.at()` to retrieve the current row index.

Transactions

In order to check, if the database driver uses a specific feature, we can use `QSqlDriver.hasFeature()` that will return a true or false value accordingly. So, we can use `QSqlDriver.hasFeature(QSqlDriver.Transactions)` to identify if the underlying database engine supports transactions. If the underlying database supports transactions, we can retrieve the commit and rollback results using the `QSqlDatabase.commit()` and `QSqlDatabase.rollback()` functions respectively. The transaction can be initiated using the `QSqlDatabase.transaction()` call. Transactions can be used to ensure that a complex operation is atomic or to provide a means of canceling a complex change in the middle.

Table and form views

This section is devoted to explaining the representation of data in a form view or a table view. But, before that we can see some examples of accessing databases through high-level model classes. The following classes are available in Qt for this purpose:

- `QSqlQueryModel`: Provides a read-only data model for SQL result sets
- `QSqlTableModel`: Provides an editable data model for a single database table
- `QSqlRelationalTableModel`: Provides an editable data model for a single database table, with foreign key support

Let us view some quick examples of each of these classes.

QSqlQueryModel

This model aims at providing a high-level interface for executing SQL queries and traversing the result set. This class is built on top of the `QSqlQuery` class and can be used to provide data to view classes, for example, `QTableView` which we are going to discuss in the forthcoming sections. A sample program using `QSqlQueryModel` is as follows:

```
model = QSqlQueryModel()
model.setQuery("SELECT fname FROM employee")
print("Names of Employee")
while i < model.rowcount() :
  print(model.record(i).value( "fname").toString())
```

In the example shown, we set the query using the `model.setQuery()` function. Once the query is set, the `QSqlQueryModel.record(int)` method is used to get individual records.

QSqlTableModel

The `PySide.QtSql.QSqlTableModel` class provides an editable data model for a single database table. As with `QSqlQueryModel`, this class also provides a high-level interface and can be used to provide data to view class. The only difference is that it allows editing of data which `QSqlQueryModel` does not support. A sample program for your reference is as follows:

```
model =  QSqlTableModel()
model.setTable("employee")
model.setFilter("age > 40")
model.setEditStrategy(QSqlTableModel.OnManualSubmit)
model.select()
```

```
model.removeColumn(0) # to remove the id column
while i < model.rowcount() :
  print(model.record(i))
```

This works as explained in our previous example. The main difference to note here is line 4. The `QSqlTableModel.setEditStarategy()` function describes which strategy we prefer to use for editing values in the database. The various options that this function can take are given as follows:

Constant	Description
`QSqlTableModel.OnFieldChange`	The changes to model will be applied immediately to the database.
`QSqlTableModel.OnRowChange`	The changes on a row will be applied when the user selects a different row.
`QSqlTableModel.OnManualSubmit`	All changes will be cached in the model until either `PySide.QtSql.QSqlTableModel.submitAll()` or `PySide.QtSql.QSqlTableModel.revertAll()` is called.

It is to be noted that to prevent inserting partial values on a row into the database, `onFieldChange` will behave like `onRowChange` for newly inserted rows. The `QSqlTableModel.setFilter()` function executes the functionality of a `WHERE` clause in SQL queries. If the model is already selected and populated, the `setFilter()` method will refilter the results. The `QSqlTableModel.setRecord()` function is used to modify a row, `QSqlTabelModel.removeRows(int)` is used to delete rows from the table.

QSqlRelationalTableModel

The `PySide.QtSql.QSqlRelationalTableModel` class serves the same purpose as `QSqlTableModel` with an additional foreign key support. An example of this model is deferred to last section after discussing the table and form views.

Table view

In the preceding sections we discussed various model classes. Now, we will look at how to present the data to the users using the `QTableView` widget. The data source for the `QTableView` widget is provided by any of the model classes. The table view is the most used view format as this represents a virtual representation of the **2D SQL** table structure. We will have a look at the code first, and then discuss its functionality.

```
import sys
from PySide.QtGui import *
from PySide.QtCore import Qt
```

```python
from PySide.QtSql import *

def initializeModel(model):
  model.setTable("employee")

  model.setEditStrategy(QSqlTableModel.OnManualSubmit)
  model.select()

  model.setHeaderData(0, Qt.Horizontal, "ID")
  model.setHeaderData(1, Qt.Horizontal, "First Name")
  model.setHeaderData(2, Qt.Horizontal, "Last Name")
  model.setHeaderData(3, Qt.Horizontal, "Age")
  model.setHeaderData(4, Qt.Horizontal, "Gender")
  model.setHeaderData(5, Qt.Horizontal, "Income")

def createView(title, model):
  view = QTableView()
  view.setModel(model)
  view.setWindowTitle(title)
  return view

def createConnection():
  db = QSqlDatabase.addDatabase('QSQLITE')
  db.setDatabaseName('sample.db')

  ok = db.open()

  if not ok:
    return False

  myQuery = QSqlQuery()

  myQuery.exec_("""CREATE TABLE employee (id INTEGER PRIMARY KEY
          AUTOINCREMENT UNIQUE NOT NULL, first_name CHAR(20) NOT NULL,
          last_name CHAR(20), age INT, sex CHAR(1), income FLOAT)""")
  myQuery.exec_("""INSERT INTO employee (first_name, last_name, age,
sex, income)
          VALUES ('Alice', 'A', 30, 'F', 5000.00)""")
  myQuery.exec_("""INSERT INTO employee (first_name, last_name, age,
sex, income)
          VALUES ('Bob', 'B', 31, 'M', 5100.00)""")
  myQuery.exec_("""INSERT INTO employee (first_name, last_name, age,
sex, income)
```

```
            VALUES ('Caesar', 'C', 32, 'F', 5200.00)""")
  myQuery.exec_("""INSERT INTO employee (first_name, last_name, age,
sex, income)
            VALUES ('Danny', 'D', 34, 'M', 5300.00)""")
  myQuery.exec_("""INSERT INTO employee (first_name, last_name, age,
sex, income)
            VALUES ('Eziekel', 'E', 35, 'F', 5400.00)""")
  return True


if __name__ =='__main__':
  try:
    myApp = QApplication(sys.argv)

    if not createConnection():
      print("Error Connecting to Database")
      sys.exit(1)

    model = QSqlTableModel()
    initializeModel(model)

    view1 = createView("Table Model - Example1", model)
    view2 = createView("Table Model - Example2", model)

    view1.setGeometry(100, 100, 500, 220)
    view2.setGeometry(100, 100, 500, 220)
    view1.show()
    view2.move(view1.x() + view1.width() + 20, view1.y())
    view2.show()
    myApp.exec_()
    sys.exit(0)
  except NameError:
    print("Name Error:", sys.exc_info()[1])
  except SystemExit:
    print("Closing Window...")
  except Exception:
    print(sys.exc_info()[1])
```

At first, we create and establish a database connection and execute the sample data in the `createConnection()` function. As discussed, a SQLite connection is created using the Qt's SQLite driver, `QSQLITE`. We set the database name in the next line. If the file exists, the database connection uses it or else creates a new file on the same name. We check the success of the connection and return false if it is not so. The sample queries are executed one-by-one in order and a true value is returned indicating the connection was successful and the data is populated. In the `initializeModel()` function, we define the properties of the model and set the display format by specifying its column headers. The `createView()` function creates a view, and returns it to the caller function. On execution, we get two table views as shown in the following screenshot. Also, note that on editing one view, the other gets updated. However, this does not update the table since we have set the edit strategy to **Manual Submit**.

Form view

The form view is useful when you want to traverse the records one-by-one and perform some modifications to it. In this section, we will see how to create a dialog form that displays the records one-by-one and also learn about how to add, edit, and delete records using the form dialog. We will use the same `employee` table created in our previous example.

We will not discuss the layout used in this program as our main aim is to discuss the form view. The model is created using `QSqlTableModel` and set the sort factor with the column `first_name` in the ascending order:

```
self.model = QSqlTableModel(self)
self.model.setTable("employee")
self.model.setSort(FIRST_NAME, Qt.AscendingOrder)
self.model.select()
```

Next, we set the mapping of the form values with the column using the `QDataWidgetMapper` class. This class is used to provide mapping between sections of a data model to widgets. The `addMapping(widget, section)` function maps the widget to the section from the model. The section represents a column if the orientation is "Horizontal", otherwise represents a row. Finally, the `toFirst()` function populates the widget from the data from the first row of the model if the orientation is "Horizontal", otherwise from the first column.

```
self.mapper = QDataWidgetMapper(self)
self.mapper.setSubmitPolicy(QDataWidgetMapper.ManualSubmit)
self.mapper.setModel(self.model)
self.mapper.addMapping(firstNameEdit, FIRST_NAME)
self.mapper.addMapping(lastNameEdit, LAST_NAME)
self.mapper.addMapping(ageEdit, AGE)
self.mapper.addMapping(genderEdit, SEX)
self.mapper.addMapping(incomeEdit, INCOME)
self.mapper.toFirst()
```

Then, we connect the buttons to their respective slots:

```
self.connect(firstButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(EmployeeForm.FIRST))
self.connect(previousButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(EmployeeForm.PREV))
self.connect(nextButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(EmployeeForm.NEXT))
self.connect(lastButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(EmployeeForm.LAST))
self.connect(addButton, SIGNAL("clicked()"), self.addRecord)
self.connect(deleteButton, SIGNAL("clicked()"),
            self.deleteRecord)
self.connect(quitButton, SIGNAL("clicked()"), self.done)
```

The slots are defined as follows:
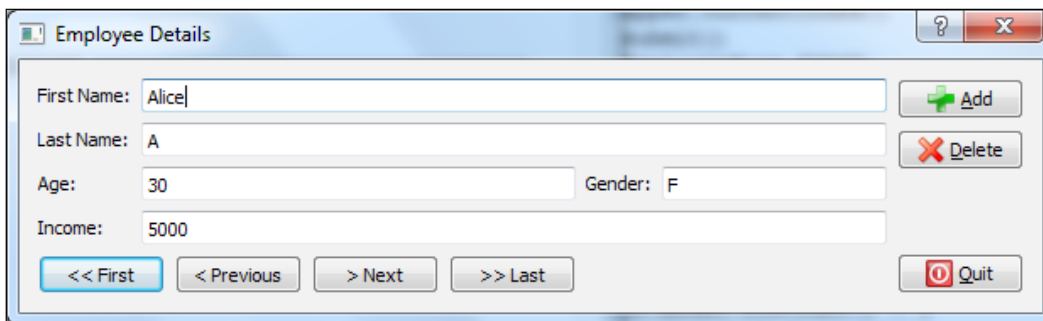
```
    def done(self, result=None):
      self.mapper.submit()
      QDialog.done(self, True)


    def addRecord(self):
      row = self.model.rowCount()
      self.mapper.submit()
      self.model.insertRow(row)
      self.mapper.setCurrentIndex(row)
```

```
def deleteRecord(self):
  row = self.mapper.currentIndex()
  self.model.removeRow(row)
  self.model.submitAll()
  if row + 1 >= self.model.rowCount():
    row = self.model.rowCount() - 1
  self.mapper.setCurrentIndex(row)


def saveRecord(self, where):
  row = self.mapper.currentIndex()
  self.mapper.submit()
  if where == EmployeeForm.FIRST:
    row = 0
  elif where == EmployeeForm.PREV:
    row = 0 if row <= 1 else row - 1
  elif where == EmployeeForm.NEXT:
    row += 1
    if row >= self.model.rowCount():
      row = self.model.rowCount() - 1
  elif where == EmployeeForm.LAST:
    row = self.model.rowCount() - 1
  self.mapper.setCurrentIndex(row)
```

The complete program can be downloaded from the code snippets that come along with this book. On execution, we will be presented with a dialog box as shown, with which we can add, edit, or delete records:

Viewing relations in table views

The main feature of relational database is its ability to relate one or more tables. One such relating feature is the use of foreign key concept where a primary key of a table is related to a column in another table. This relation can be easily exhibited using `QRelationalTableModel`. In order to explain this, we create three tables that are connected to each other. The schema is defined as follows:

```
CREATE TABLE employee (id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT
NULL, name VARCHAR(40) NOT NULL, department INTEGER, branch INTEGER)
CREATE TABLE department (id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE
NOT NULL, name VARCHAR(20) NOT NULL, FOREIGN KEY(id) REFERENCES
employee)
CREATE TABLE branch (id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT
NULL, name VARCHAR(20) NOT NULL, FOREIGN KEY(id) REFERENCES employee)
```

If we use `QSqlTableModel`, we will get a view as given in the following screenshot:
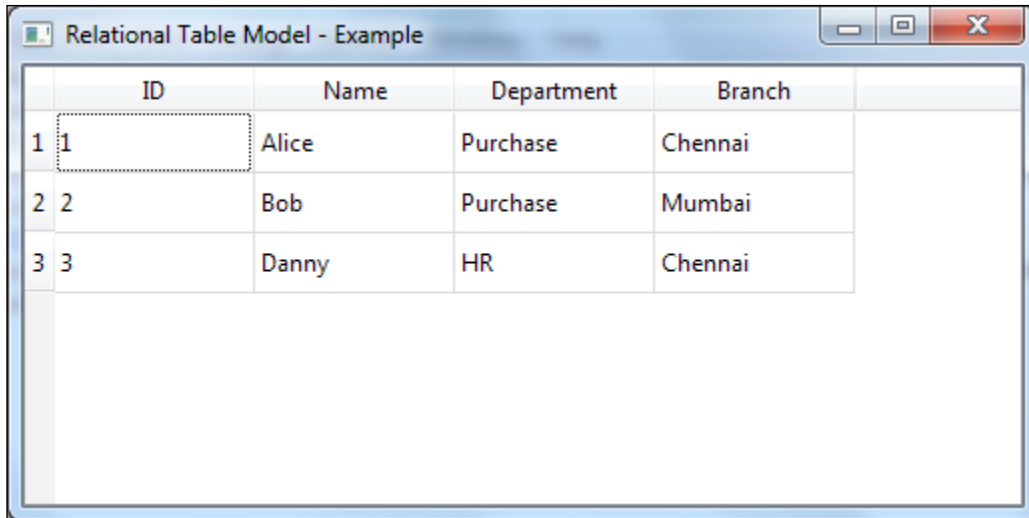
By using relational table model, we can reference the department and branch into their relations as follows:

```
model.setRelation(2, QSqlRelation("department", "id", "name"));
model.setRelation(3, QSqlRelation("branch", "id", "name"));
```

This code will set the relations of department and branch column to their respective tables along with the column value that has to be displayed on the view. On setting the relations, the view will be modified as shown in the following screenshot where the ID's are resolved into their respective names:

Thus, the relational model is so helpful in exhibiting relational databases.

Summary

Most of the real-time applications have to deal with databases for data storage and retrieval. Thus, it is more important for a GUI developer to know about database interaction. Qt is supplemented with many built-in database drivers which we can use to connect to databases and perform desired operations. Also, with its wide variety of model and view classes, it becomes very easy for any GUI programmer to interact with databases.

Resources

This appendix brings to you some resources that are available online for
PySide application development.

The PySide documentation wiki page

The PySide documentation wiki page has a lot of information and links to various
PySide resources. It hosts a number of links to tutorials and reference manuals that a
beginner programmer can start with. It can be accessed at `http://qt-project.org/
wiki/Category:LanguageBindings::PySide`.

API reference manuals

The API reference manual documents the usage of various available classes for PySide.
The PySide class reference documentation is automatically generated from the original
Qt documentation for C++, and some parts are tuned to fit the Python world.

- PySide 1.0.7 reference: `http://srinikom.github.io/pyside-docs/`
- PySide 1.1.0 reference: `https://deptinfo-ensip.univ-poitiers.fr/ENS/
  pyside-docs/index.html`

Tutorials

A set of basic and advanced tutorials can be found at this page:
`http://qt-project.org/wiki/PySide_Tutorials`

Community support

Often all open source software is supported through the community forum where the developers dwell together to help and support others. The various community supports that are available are given as follows. Developers can contact these forums to get support for PySide development and contribution:

- Active mailing list: `http://lists.qt-project.org/mailman/listinfo/pyside`
- Archive mailing list: `http://dir.gmane.org/gmane.comp.lib.qt.pyside`
- Forum on Qt-project: `http://qt-project.org/forums/viewforum/15/`
- Internet relay chat: `#pyside` channel at `https://webchat.freenode.net/`

Apart from these resources there are finer examples available at Github for reference.

Index

event() function 61
event handlers
  re-implementing 63-66
event loop 62
event management
  about 61
  event loop 62
event() method 65
event processing, Qt
  about 63
  event filters, installing 66, 67
  event handlers, re-implementing 63-66
  notify() function, re-implementing 67
exception handler 16
exception handling
  about 15
  as practice 16-18
executedQuery() function 105

F

Fminer 6
form view 111-113
forum on Qt-project
  URL 118
frameGeometry() function 30

G

Generator Runner 11
graphics 77, 78
graphics view classes
  QGraphicsItem 77
  QGraphicsScene 77
  QGraphicsView 77
GUI 6, 7

I

initializeModel() function 111
installation, event filters 66, 67
installation, PySide
  on Linux 8, 9
  on Mac OSX 8
  on Windows 8
Internet relay chat
  URL 118
IRC (Internet Relay Chat) 5

K

keyPressEvent() function 64, 65
keyPress() function 66
keyReleaseEvent() function 65

L

layout containers 50
layout management approaches
  about 49
  absolute positioning 49
  layout containers 50
LCDNumber 34
LGPL Version 2.1 license 6
Linux
  PySide, building on 10
  PySide, installing on 8, 9
Lucas Chess 6

M

Mac OSX
  PySide, building on 12
  PySide, installing on 8
main window
  creating 38, 39
matplotlib 6
MDI
  about 55
  implementing 98, 99
MDI application
  example 98
menu 43
Menu bar
  about 43
  adding, to application 44
  menu list, adding 45, 48
menu list
  adding, to menu bar 45, 48
MIME type 71
Multiple Document Interface. *See* **MDI**

N

named binding
  example 104

PACKT PUBLISHING

Thank you for buying
PySide GUI Application Development

About Packt Publishing

Packt, pronounced "packed", published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Spring Python 1.1

ISBN: 978-1-84951-066-0   Paperback: 264 pages

Create powerful and versatile Spring Python
applications using pragmatic libraries and
useful abstractions

1. Maximize the use of Spring features in
    Python and develop impressive Spring
    Python applications.

2. Explore the versatility of Spring Python by
    integrating it with frameworks, libraries,
    and tools.

3. Discover the non-intrusive Spring way of wiring
    together Python components.

4. Packed with hands-on-examples, case studies,
    and clear explanations for better understanding.
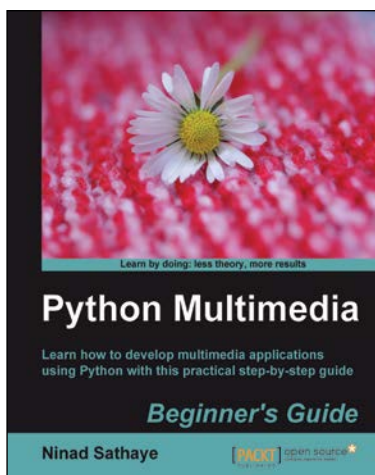
Python Multimedia
Beginner's Guide

ISBN: 978-1-84951-016-5   Paperback: 292 pages

Learn how to develop multimedia applications using
Python with this practical step-by-step guide

1. Use the Python Imaging Library for digital
    image processing.

2. Create exciting 2D cartoon characters using
    Pyglet multimedia framework.

3. Create GUI-based audio and video players using
    QT Phonon framework.

4. Get to grips with the primer on GStreamer
    multimedia framework and use this API for
    audio and video processing.

Please check **www.PacktPub.com** for information on our titles

Expert Python Programming

ISBN: 978-1-84719-494-7   Paperback: 372 pages

Best practices for designing, coding, and distributing your Python software

1. Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions.

2. Apply object-oriented principles, design patterns, and advanced syntax tricks.

3. Manage your code with distributed version control.

4. Profile and optimize your code.
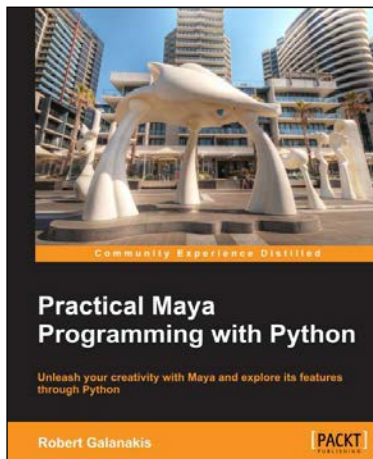
Practical Maya Programming with Python

ISBN: 978-1-84969-472-8   Paperback: 321 pages

Unleash your creativity with Maya and explore its features through Python

1. Create and customize UIs using standard tools and libraries.

2. Understand how Maya can leverage advanced Python features.

3. Make your tools, and Maya itself, fully automatable.

Please check **www.PacktPub.com** for information on our titles