

PyQt Whitepaper

www.riverbankcomputing.com

Abstract

This whitepaper provides an introduction to Agile programming using PyQt—the Python bindings for the Qt[®] application development framework. The whitepaper briefly makes the case for the Agile programming approach and for the use of the very high level dynamic language, Python. It also presents some of the key benefits available from the Qt application development framework and shows how the combination of Qt and Python—PyQt, provides an ideal combination for rapidly creating powerful and flexible GUI applications. PyQt applications can run on any platform that has PyQt, Python, and the Qt libraries installed, simply by copying the application—and with no source code changes necessary. (It is also possible to create stand-alone executables.) The currently supported platforms include all versions of Windows from Windows 98 to Vista, and most Unixes and Unix clones that run X11, including Mac OS X, Linux, Solaris, and HP-UX. Using PyQt, developers can write applications with native look and feel on all these platforms, without having to waste their time on ironing out platform differences or filling in gaps, since all these details are handled automatically by PyQt.

Contents

1. Executive Summary	2
2. The Case for Agile Development	2
3. The Case for Python	3
4. The Case for Qt	5
5. The Case for PyQt	6
5.1. An Example Application	7
5.2. An Example Custom Widget	10
6. Success Stories	12
7. Conclusion	13

1. Executive Summary

Many organisations need to develop GUI applications—some as in-house tools, and others for their customers to buy and deploy. Two problems have traditionally beset GUI programming. First, the ability to write applications that will run on all the required platforms. Even those who only deploy on Windows are faced with a variety of APIs and version differences that can end up consuming a lot of developer time. And nowadays, servers are as often Linux-based as Windows-based, while both Mac OS X and Linux are making significant inroads in the desktop space. Second, developing GUI applications can be very time consuming and technically challenging using some of the languages and libraries that are currently available. This can narrow the range of people who can usefully be involved in developing GUI applications, a significant loss for software aimed at scientists, engineers, doctors, and other professionals whose involvement could make products more suitable for their peers.

PyQt offers a solution to both the problems just described, and also brings many benefits that are not available from other technologies. In this whitepaper we will make the case for the Agile development approach to GUI application development, and then make the case for the two foundations on which PyQt is built—the Python programming language and the Qt[®] application development framework. We will then make the case for PyQt itself, and follow this with two examples, one showing code extracts from a simplified PyQt application, and the other showing the complete code for a very basic custom widget. These examples are provided to help programmers and other readers with programming knowledge to get a more direct feel for what PyQt code looks like and for how it works. Then we will discuss at just a few of the applications that have been developed using PyQt to give a flavour of what has been done and a taste of what is possible.

2. The Case for Agile Development

Back in the 1980s a new development paradigm was developed—RAD (Rapid Application Development). The purpose of RAD was to increase the speed and quality of development at a time when development processes were slow and cumbersome. Applications often took so long to develop that the requirements changed before the application was completed.

Some of the general ideas that RAD introduced were:

- Deadlines are more important than the feature set.
- The approach to development should be multi-disciplinary—analysis, design, and development occurring at the same time.
- The quality of the code produced is not the most important goal.
- Higher costs are acceptable in order to use tools that will reduce development time.
- Multiple prototypes will be produced, and they are intended to be discarded.

While high-end CASE (Computer Aided Software Engineering) tools were developed as a response to the popularity of RAD, at the other end of the spectrum, Microsoft probably had the most practical impact on RAD with the introduction of Visual Basic. This tool not only helped with RAD, it facilitated the production of simple applications by technical professionals with little or no prior knowledge of software development.

RAD, as originally defined, fell out of favour due to a number of limitations, in particular:

- Due to the reduced focus on features, the resulting applications tended to provide less customer satisfaction.
- The throwing away of prototypes was considered to be wasteful.
- Applications were difficult to scale since their architecture had evolved from prototypes rather than being the result of careful design analysis.

Agile software development represents a successor approach to RAD. Agile development is characterized by:

- Responsiveness to change.
- Increased speed to market.
- Lower cost.
- Greater delivered value.
- Iterative and incremental development—working software is the measure of progress.

Agile development focuses on iterative development with working software (versus prototypes) delivered on a regular basis, typically every few weeks. These iterations cover the entire spectrum of software development: planning, requirements analysis, design, coding, testing, and documentation. Unlike more traditional methods, the emphasis on each of these iterations is adapting to the problems and feedback revealed and reported regarding the previous iteration.

Practically speaking, agile development occupies a middle ground between RAD and more traditional development practices. It addresses the weaknesses of RAD while leveraging the strengths of more traditional methods, but without their overhead.

3. The Case for Python

As the popularity of dynamic languages increased with non-professional software developers, their usage evolved from prototyping/RAD tools into languages of choice for many applications. They have also proved very popular with software professionals, since they improve development speed in two ways. First, dynamic languages don't force programmers to go through a time-consuming edit-compile-link-run sequence—new code can simply be edited and run. Second, dynamic languages are usually higher level, that is, programmers can express what they want using less code, so they are more productive.

The most popular RAD language, Visual Basic, brought significant disadvantages as well as advantages. In particular, it can be difficult to build large maintainable applications using Visual Basic, and it is also targeted at a single platform, Windows. With the introduction of the .NET toolkit, Microsoft introduced Visual Basic.NET. This language differs from Visual Basic in some important ways, largely due to the fact that its underlying execution environment was designed to support non-dynamic programming languages such as C#. Some programmers have found it difficult to move old code to the new environment, and this has led to more interest in exploring alternatives.

As agile development practices continue to gain traction, software developers have begun to turn to dynamic languages to meet their needs. Dynamic languages are a new, more powerful subset of what were originally known as “scripting” languages. A general way of describing the difference is that a scripting language supports simple ways of specifying tasks to be performed, while dynamic languages borrow features from traditional compiled programming languages and provide a robust and dynamic execution environment. Another way to view the role and power of dynamic languages

is as “integration” or “application” languages; they can be used to quickly and easily integrate diverse off-the-shelf functionality into a single application.

In the same way that agile software development techniques are a compromise between RAD and more traditional software development methodologies, dynamic languages inhabit a middle ground between scripting languages and strongly typed/compiled programming languages. While traditional programming languages are well suited to complex and lower level tasks such as framework development, dynamic languages are more appropriate for application development. Using dynamic languages also makes it much easier and quicker to adapt to changes, such as those driven by user requirements, or by external forces such as Government regulations.

Python is an open source dynamic language that was created by Guido van Rossum in 1990 (www.python.org). Since that time, it has matured and grown in popularity to be one of the top ten most referenced computer languages on the Internet (see www.tiobe.com/tpci.htm). The case can be made that Python is an “agile” dynamic language; here are some of its highlights:

- The language and its extensive standard library are intuitive to use and easy to learn.
- Beginners can very quickly be productive with Python.
- Experts can make use of Python’s many advanced features, such as partial function application, metaprogramming, and threading—there are no artificial limits. And all Python objects are first class, whether they are built into the language or custom made by Python programmers.
- The language has a simple yet elegant object-oriented design.
- Python code is easy to read as well as easy to write—and Python is so high level that it can almost be considered “executable pseudo-code”.
- The language is highly scalable—it is used for projects varying in size from hundreds, to thousands, to tens of thousands, to hundreds of thousands of lines of code.
- It is very suited to rapid development and makes refactoring easy.
- Programs are portable across platforms and (especially when using PyQt) can be deployed on all platforms without requiring changes.
- Has a very extensive standard library, with many additional libraries and frameworks available, rivaling .NET in functionality.
- Is easily extensible, by writing custom libraries in Python, or by writing extensions in other languages such as C and C++.
- Programs are concise—a Python solution is about 50% the size of a comparable C++ solution.
- Increasingly being used in academia to teach programming—and not just for computer science students—so Python skills and knowledge are becoming more readily available.

The development cycle also tends to be both easier and faster with Python. Like many dynamic languages, Python is interpreted from automatically compiled byte-code. This process is invisible to programmers, resulting in a fast edit-run cycle versus the traditional edit-compile-link-run cycle. Additionally, since Python is a managed execution environment, all run-time errors show up in a standardized exception format. Combined with automatic memory management (garbage collection), this prevents an entire class of difficult-to-debug problems encountered by traditional programming languages such as C and C++.

Integration is easily accomplished with Python; extension modules are usually implemented in either Python, C or C++. Using tools such as SIP (www.riverbankcomputing.com/software/sip) and Boost.Python (www.boost.org) it is relatively straightforward to create an extension module that encapsulates an existing C or C++ library. This is also useful in situations where performance sensitive algorithms need to be implemented outside Python’s managed execution environment. And for numeric processing the NumPy and SciPy libraries (www.scipy.org) offer considerable additional functionality, including compact and fast n -dimensional arrays.

Application developers often have to handle the complexities of the packaging and distribution of their applications. There are tools available for turning Python applications into distributable executables. These tools enable applications to be distributed without the source code and without the target system needing to have Python or the libraries used by the application installed on the target machines. PyInstaller (pyinstaller.python-hosting.com) and py2exe (www.py2exe.org) are two of the more popular ones.

While Python is an open source language and freely available under a very liberal license, there are commercial offerings available for those organizations that require it. These provide the investment protection that many companies look for when choosing a development technology. For example, ActivePython (www.activestate.com) is a commercially supported distribution of Python and commercial IDEs such as WingIDE (www.wingware.com) are also available.

4. The Case for Qt

Qt[®] is a cross-platform application development framework developed by the Qt Software division of Nokia Corporation (www.qtsoftware.com). Users of Qt include Adobe, Google, IBM, Sharp, and Siemens. Adobe Photoshop Album and Google Earth are just two examples of widely used cross-platform Qt applications. Qt is the foundation on which Linux's answer to Windows—KDE (the “K” Desktop Environment)—is built. Qt is implemented in C++ and is fully object-oriented—it provides more than 600 classes, all with sensible defaults and useful functionality out of the box, and all able to be customized and subclassed to meet programmer requirements.

Qt uses Unicode throughout and is supplied with tools to make translation and localisation as straightforward as possible. In addition, Qt comes with *Qt Designer*, a GUI tool itself written in Qt that provides a means of designing dialogs visually. *Qt Designer* fully understands Qt's sophisticated yet easy to use layout system, and creates dialogs that automatically adjust to their content, rather than using the fixed sizes and positions that can be so frustrating to users of Visual Basic applications. *Qt Designer* generates XML files that are converted into C++, a step that is taken care of automatically by the build process. Although separating code and presentation is preferred by many development teams, Qt's layout system is designed for hand-coding, so using *Qt Designer* is not necessary for those who prefer all-code development.

One key innovation that Qt introduced is the signals and slots mechanism. Now widely copied in other GUI libraries, this mechanism provides an abstraction for handling events. While it is easy to get all the details of mouse clicks and key presses in Qt through its low-level event handling system, more often what programmers really care about is what the user actually wants to do, not how they asked to do it. For example, no matter whether the user typed Ctrl+S or clicked a Save toolbar button, or clicked a File→Save menu option, in almost every case, all that really matters is that they want to save their work. The signals and slots mechanism provides the high level view that makes it easy to handle the “save” action—or any other action—no matter how it was invoked.

Signals and slots is a generalisation of the Observer design pattern that provides type-safe communication between objects. When an object changes state or has some other significant event occur (such as when a button is clicked), the object emits a signal (e.g., “clicked”). If we are interested in an occurrence, we can connect to the relevant object and to the type of signal the object sends that we are interested in—for example, we can connect a particular button's “clicked” signal to the method we want called when the button is clicked. Neither the emitting object (e.g., the button), nor the receiving object (e.g., the object whose method will be called), need have any knowledge of the other. All that must be known is that the one emits a suitable signal and that the other can connect to the signal. Signals can be emitted with data, and the data is passed in a type-safe way, avoiding the risks and limitations of the callbacks that most older frameworks employ.

Qt comes with a large and comprehensive set of widgets (“controls” and “containers” in Windows-speak) from simple line edits to calendars, tables, tree views, all the way up to a web browser widget that supports HTML, XHTML, CSS, JavaScript, HTML canvas, and AJAX. In fact, many applications are built purely using Qt's standard widgets. Nonetheless, one of the outstanding features of the framework is its support for creating custom widgets. Custom widgets can be created by subclassing existing widgets, or they can be created from scratch with the programmer exercising complete con-

trol over their appearance and behaviour. And Qt offers a lot more than GUI widgets; here are some of its other highlights:

- Excellent 2D graphics support, including semi-transparency, antialiasing, a floating-point coordinate system, gradients, painter paths, and automatic double-buffering (off-screen rendering) for flicker-free updates. 3D graphics is supported using OpenGL[®], and support for all the most popular bitmap image formats, including PNG, JPEG, GIF, and BMP, and support for reading and writing SVG vector images is built in.
- Audio and video playback using the Phonon library integration.
- GUI testing with the QTest module; this allows tests to perform mouse actions and key presses to mimic users.
- Support for wizards (“assistants” in Mac-speak)—these are dialogs that take users step-by-step through tasks; they are most frequently used by installers.
- A model/view framework for MVC (Model-View-Controller)-style programming to support the separation of data from how it is visualised and edited. This framework can be used with your own data structures, or with Qt’s database classes with which it is fully integrated. The uniform database API works the same no matter what platform Qt runs on and no matter which database Qt is used with.
- Rich text viewing and editing using HTML syntax that can be written in source code. Rich text (including embedded images) can be printed or saved in PDF format.
- Libraries for parsing and writing XML, including conventional DOM and SAX parsers, and some very fast Qt-specific XML classes.
- Powerful and flexible networking classes with support for major high level protocols such as HTTP and FTP, as well as low level TCP and UDP classes. These classes can be used asynchronously so as not to block the user interface, or synchronously in non-GUI threads.
- Solid and efficient cross-platform support for threading—the Qt library is multithreaded by default, and provides both low level threading classes and a high level concurrency abstraction to make threaded programming as simple as possible.
- Native look and feel on all platforms including Vista, and with full support for programmer customization, for example using simple CSS (Cascading Style Sheet)-type styles, or by creating style subclasses for extremely detailed control.

Unlike some other GUI libraries, Qt was built using an object-oriented design from the very beginning. After more than 15 years of commercial development and use, Qt is now a very mature high quality product with unmatched breadth, that presents a consistent object-oriented API that minimises learning time and makes Qt programming and maintenance as easy, pleasant, and productive as possible.

Qt is dual licensed—the LGPL version can be used for most purposes, and the commercial licensed version can be used when the conditions of the LGPL are considered too restrictive. Qt is used for both in-house software development, and to produce commercial products sold on the open market. In fact, some companies that use Qt do not disclose their use because they consider that Qt gives them a commercial advantage that they don’t want their competitors to be aware of. For a much more detailed technical overview, running to over 70 pages, see Qt Software’s Qt whitepaper (www.qtsoftware.com/files/pdf).

5. The Case for PyQt

PyQt is a product of Riverbank Computing Limited, an independent software development company that specializes in some key open source software technologies. PyQt is a *binding*, that is, a set of libraries that make the C++/Qt application development framework’s libraries accessible to Python programmers. PyQt presents a very *Pythonic* (good Python style) API to Python programmers, which

makes learning PyQt as easy as possible and means that good Python programming style is also good PyQt programming style.

PyQt programmers can create all their programs purely in code, or they can use *Qt Designer* to visually create dialogs; PyQt comes with a tool for converting *Qt Designer* XML files into Python code. *Qt Designer* allied to Python's fast edit-run cycle creates an agile development environment—or a rapid prototyping toolset for applications that are intended to be written in C++.

Almost the entire Qt library of over 600 classes is available to PyQt programmers through consistent and easy to use Python APIs that closely match the underlying C++/Qt APIs. All Qt's widgets are available and custom widgets can be created from subclassing or from scratch using the same techniques used for C++/Qt programming. All the Qt highlights mentioned earlier are available in PyQt, including 2D and 3D graphics support, the model/view framework, HTML editing and viewing, XML parsing and writing, networking, threading, and look and feel customization.

PyQt is supplied with source code, and a dual licensing scheme allows it to be licensed under the GPL and a commercial license. Unlike Qt, PyQt is not licensed under the LGPL.

5.1. An Example Application

The ListKeeper application is a very simple conventional main-window style application with a menu bar, toolbars, and a central widget. A screenshot is shown in Figure 1. We will show some code extracts to give a flavour of PyQt programming.

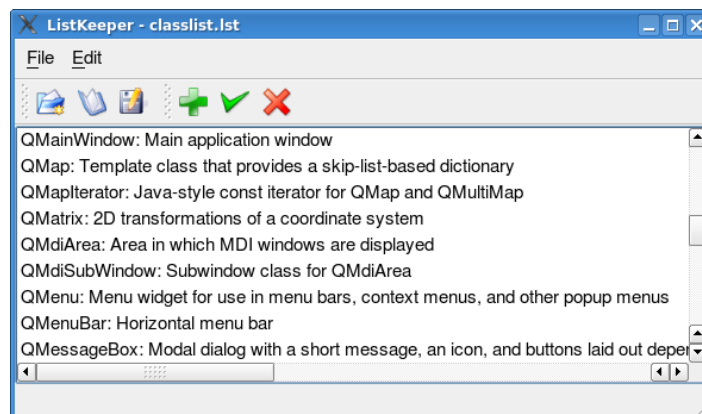


Figure 1. The ListKeeper Application

We have imported the PyQt classes using these two lines:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

This makes the code shorter and easier to read, but many developers don't like using this syntax since it brings large numbers of names into their namespace. This isn't really a problem with PyQt though, because almost every name imported begins with "Q". However, in practice, most developers write `from PyQt4 import QtCore, QtGui`, so for example, instead of writing `self.listWidget = QListWidget()`, they write `self.listWidget = QtGui.QListWidget()`.

Each form (window) in the application is represented by a class. Here is the start of the MainWindow class's initializer:

```
class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)

        self.filename = None
```

```

self.dirty = False
self.listWidget = QListWidget()
self.setCentralWidget(self.listWidget)

```

In this simple example the central widget is just a list widget; in more sophisticated applications we might use a table widget, or a custom widget.

Here is the code for creating some of the file-related actions:

```

fileNewAction = self.createAction("&New...", self.fileNew,
    QKeySequence.New, "filenew")
fileOpenAction = self.createAction("&Open...", self.fileOpen,
    QKeySequence.Open, "fileopen")
fileSaveAction = self.createAction("&Save", self.fileSave,
    QKeySequence.Save, "filesave")

```

Each action is represented by an object of type `QAction`. These objects hold a menu text, a method to be called when they are invoked, a keyboard shortcut, and an icon. They can also have additional data such as tooltips and status tip texts. Actions are added to menus and toolbars, for example:

```

fileMenu = self.menuBar().addMenu("&File")
fileMenu.addAction(fileNewAction)
fileMenu.addAction(fileOpenAction)
fileMenu.addAction(fileSaveAction)

fileToolbar = self.addToolBar("File")
fileToolbar.addAction(fileNewAction)
fileToolbar.addAction(fileOpenAction)
fileToolbar.addAction(fileSaveAction)

```

Once an action has been added to a menu or toolbar (or both), its keyboard shortcut becomes active. So to save their data the user can type `Ctrl+S`, or click the Save toolbar button, or click the File→Save menu option—and whichever they use will lead to the `fileSave()` method being invoked.

```

self.statusBar().showMessage("Ready...", 5000)
self.setWindowTitle("ListKeeper - Unnamed List")

```

At the end of the initializer we set an initial message for the status bar (with a 5 second timeout after which it will be cleared), and the window's title.

If the user chooses to add a new item they invoke the “add” action (the code for which isn't shown, but is just the same as for the other actions), and the `editAdd()` method will be invoked.

```

def editAdd(self):
    form = AddForm(self)
    if form.exec_():
        if form.addAtEnd:
            self.listWidget.addItem(form.text)
        else:
            self.listWidget.insertItem(0, form.text)
        self.dirty = True

```

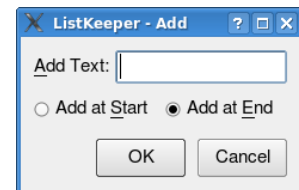


Figure 2. The Add Dialog

This method creates an instance of the custom `AddForm` class and calls `exec_()` on it. The `exec_()` method shows the form modally and returns `True` if the user accepted the dialog (e.g., by pressing OK). Because it is so simple we have created the form entirely in code. First we create the widgets, then we lay them out, and then we create signal and slot connections to get the behaviour we want.

```

class AddForm(QDialog):

    def __init__(self, parent=None):

```

```

super(AddForm, self).__init__(parent)

self.textLineEdit = QLineEdit()
label = QLabel("&Add Text:")
label.setBuddy(self.textLineEdit)
self.addAtStartRadioButton = QRadioButton("Add at &Start")
self.addAtEndRadioButton = QRadioButton("Add at &End")
self.addAtEndRadioButton.setChecked(True)
buttonBox = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel)

```

The form requires a line edit, a label, a couple of radio buttons, and OK and Cancel buttons. The code for creating most of these is trivial. The ampersands in the labels are used to indicate accelerator keys, so `&Add Text:` will appear as `Add Text`. The `setBuddy()` call is used to tell the label that if its accelerator is invoked (`Alt+A`), then focus should be given to the line edit. PyQt is smart enough to make the radio buttons work exclusively (that is, if one is selected the other is automatically unselected); and also provides group boxes for more explicit control when this is needed.

Buttons can be created individually using the `QPushButton` class, but here we have used a `QDialogButtonBox`. This has the advantage that we don't have to worry about button order (which is different on Windows and Mac OS X for example), since the `QDialogButtonBox` will show the buttons on the correct order for the platform on which the application is run.

```

topLayout = QHBoxLayout()
topLayout.addWidget(label)
topLayout.addWidget(self.textLineEdit)
middleLayout = QHBoxLayout()
middleLayout.addWidget(self.addAtStartRadioButton)
middleLayout.addWidget(self.addAtEndRadioButton)
layout = QVBoxLayout()
layout.addLayout(topLayout)
layout.addLayout(middleLayout)
layout.addStretch()
layout.addWidget(buttonBox)
self.setLayout(layout)

```

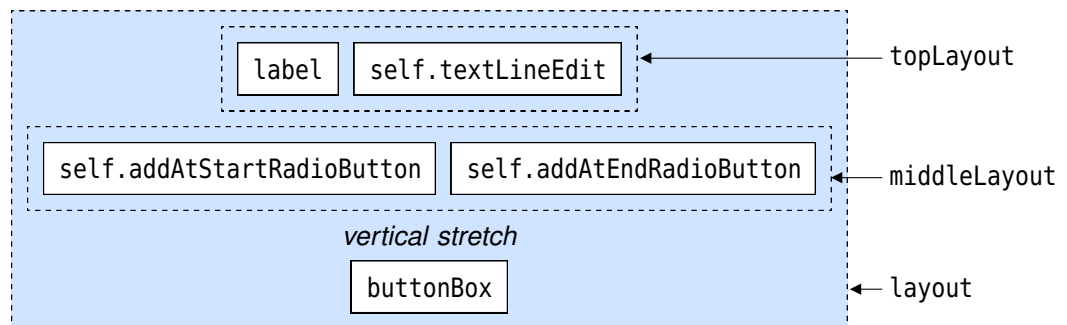


Figure 3. The Add Dialog's Layout

Having created all the widgets, we need to lay them out. PyQt provides three layout managers, the horizontal and vertical ones used here, and a grid layout. The layout system is remarkably smart, so it is not uncommon to have code like this where we simply add widgets and layouts to layouts without needing to do any manual tweaking—although it is possible to tweak if necessary. The layout is shown schematically in Figure 3. Notice in the screenshot (Figure 2) how PyQt automatically gives the label exactly the amount of space that it needs, and stretches the line edit to occupy the remaining width to make it as wide as possible.

```

buttonBox.accepted.connect(self.accept)
buttonBox.rejected.connect(self.reject)

self.setWindowTitle("ListKeeper - Add")

```

At the end of the initializer we connect two of the button box’s signals to the methods we want invoked. Unlike C++/Qt where signals can only be connected to methods that are specially designated (“slots”), in PyQt signals can be connected to any callable.

We only need to reimplement the `accept()` method because the inherited `reject()` method already does what we want—closes the dialog and returns `False` from `exec_()`. And as usual, we finish the initializer by setting the window’s title.

```

def accept(self):
    self.text = self.textLineEdit.text()
    if self.text.isEmpty():
        super(AddForm, self).reject()
    else:
        self.addAtEnd = self.addAtEndRadioButton.isChecked()
        super(AddForm, self).accept()

```

If the user clicks OK, we close the dialog. If the user didn’t add any text we call `QDialog.reject()` which will return `False` from `exec_()`; otherwise we call `QDialog.accept()` which makes `exec_()` return `True`. It would have added just a few more lines to disable the OK button whenever there was no text in the line edit, in which case `accept()` would only be called if there was text, but we have tried to keep the example as simple as possible while still being illustrative.

5.2. An Example Custom Widget

Both the `MainWindow` and `AddForm` classes discussed in the previous subsection are custom widgets. The `AddForm` widget’s behaviour and appearance has been created by instantiating and laying out widgets inside it and by making signal–slot connections, a process that is typical of how custom dialogs are created all in code using PyQt. If we had used *Qt Designer* we could have done the layout visually—this can be much faster, especially for those new to PyQt programming, and also makes it easy to change designs, usually without needing to change the source code.

It is possible to create custom widgets from scratch by creating a `QWidget` subclass and reimplementing the low-level event handlers. In such cases it is usually sufficient to reimplement the `mousePressEvent()` method (and sometimes also the `mouseMoveEvent()` and `mouseReleaseEvent()` methods) to respond to mouse clicks, the `keyPressEvent()` method to handle key presses, and the `paintEvent()` method to draw the widget’s appearance however we like. It is also common to reimplement the `sizeHint()` method to help the layout system understand the widget’s layout requirements, and sometimes the `minimumSizeHint()` and `resize()` methods.

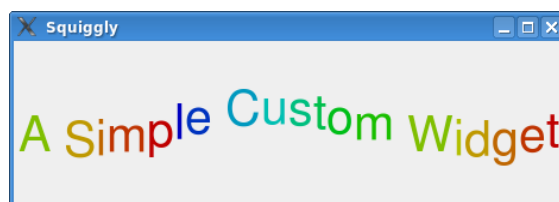


Figure 4. The Squiggly Widget

In this section we will briefly review the code for a very simple custom widget. This widget shows a line of animated text—the text moves with a wave-like motion and its colours change to produce a striking effect that the screenshot in Figure 4 cannot do justice to.

```

class Squiggly(QWidget):

```

```
textChanged = pyqtSignal(str)

def __init__(self, parent=None):
    super(Squiggly, self).__init__(parent)

    font = self.font()
    font.setPointSize(font.pointSize() + 20)
    self.setFont(font)
    self.text = QString("Squiggly")
    self.step = 0;
    self.setWindowTitle("Squiggly")
    self.resize(360, 150)
    self.timer = QTimer()
    self.timer.start(60, self)
```

The initializer is quite different from the ones we have seen before. There is no central widget with actions, nor other widgets in a layout. Instead this custom widget will create its own appearance and provide its own behaviour. The initializer is used to set the widget's font size to be much larger than normal, to give it an initial text to display, to give it an initial size, and to start off a 60 millisecond timer.

The widget's appearance and behaviour is completely defined by four low-level event handlers. We will look at them all, starting with the timer event handler.

```
def timerEvent(self, event):
    if (event.timerId() == self.timer.timerId()):
        self.step += 1
        self.update()
    else:
        super(Squiggly, self).timerEvent(event)
```

Whenever the timer times out a timer event occurs. By reimplementing the widget's `timerEvent()` method we can intercept such events and either handle them, or pass them on for PyQt to handle for us. If the timer event was caused by the widget's own timer, we increment the step rate (used in drawing the widget), and call `self.update()`—this call schedules a paint event, so that the widget can repaint its contents using the new step size.

```
def paintEvent(self, event):
    sines = (0, 38, 71, 92, 100, 92, 71, 38, 0, -38, -71, -92, -100, -92, -71, -38)
    fm = QFontMetrics(self.font())
    x = (self.width() - fm.width(self.text)) / 2
    y = (self.height() + fm.ascent() - fm.descent()) / 2
    color = QColor()
    painter = QPainter(self)

    for i in range(self.text.size()):
        index = (self.step + i) % 16
        color.setHsv((15 - index) * 16, 255, 191)
        painter.setPen(color)
        painter.drawText(x, y - ((sines[index] * fm.height()) / 400), self.text[i])
        x += fm.width(self.text[i])
```

Whenever a paint event occurs, this method is called. A paint event is always issued just before a widget is shown for the first time, and in this widget, whenever a timer event occurs. The `x` and `y` coordinate step sizes are calculated based on the widget's font's font metrics. A `QPainter` is used to draw on the widget's surface—PyQt automatically double-buffers to avoid flicker. In this case we only use the `QPainter.drawText()` method, but a whole range of methods for drawing text, standard shapes, arbitrary polygons and polylines, gradients, and painter paths (rather like PostScript strokes) are available.

```
def keyPressEvent(self, event):
    if event.key() in (Qt.Key_Q, Qt.Key_X, Qt.Key_Escape):
        self.close()
    else:
        super(Squiggly, self).keyPressEvent(event)
```

If the user presses a key the key press handler is called. Here we check for Q, X, or Esc, and if any of these was pressed we close the widget. Otherwise we pass the keypress on to PyQt to handle.

```
def mousePressEvent(self, event):
    text, ok = QDialog.getText(self, "Squiggly - Set Text", "Text:",
                               QLineEdit.Normal, self.text)
    if ok and not text.isEmpty():
        self.text = text
        self.update()
        self.textChanged.emit(self.text)
```

If the user clicks the widget we pop up a dialog asking them to enter the text to be displayed and use the current text as the default. If they enter a new text and press OK in the dialog we change the widget's text accordingly and call `self.update()` to schedule a paint event. We also emit a custom "textChanged" signal with the new text as argument. Programmers using the Squiggly class are free to connect to this signal, or to ignore it. In this case we have chosen not to pass the mouse event on to PyQt, even if we don't handle it.

The small initializer and the event handlers are sufficient to create the Squiggly custom widget. And we can test the widget without having to embed it in an application, simply by adding some code at the end of its .py file:

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    squiggly = Squiggly()
    squiggly.show()
    app.exec_()
```

Applications would normally do `import squiggly` and create an instance of the custom widget with `squiggly.Squiggly()`, but if the widget's module file is run stand-alone, these lines execute and allow us to test the widget.

Creating complex cross-platform custom widgets is perfectly possible with PyQt, although naturally this normally involves creating rather more sophisticated paint event handlers that we have used here, and reimplementing at least one size hint method to make the widget work harmoniously with the layout system.

6. Success Stories

PyQt's commercial adoption has grown significantly since the introduction of PyQt4, the Python bindings for the Qt 4 C++ libraries, with customers from around the world. PyQt is also used to create many open source GUI applications.

To take an initial example, Yadda is a communication application developed by Verscend Technologies, a company specializing in software development using Python and Qt. Yadda is a client/server application for teams that solves real-time communication and logistics problems. Verscend report that: "this application consists of approximately 15 000 lines of Python code the majority of which was delivered in a 3 month time-frame with a remarkably low defect rate."

Another example is Italian consulting company Develer (www.develer.com) which provides B2B solutions by providing outsourced development and technical support for its customers. They report: "In the early days of the desktop development team, Develer discovered that PyQt made rapid development and deployment to customers possible, both for throw-away prototypes and production quality

solutions.” Two of their most successful PyQt projects were a 2D CAD system for fashion designers using OpenGL, and a fully internationalised medical diagnosis application, with a database backend.

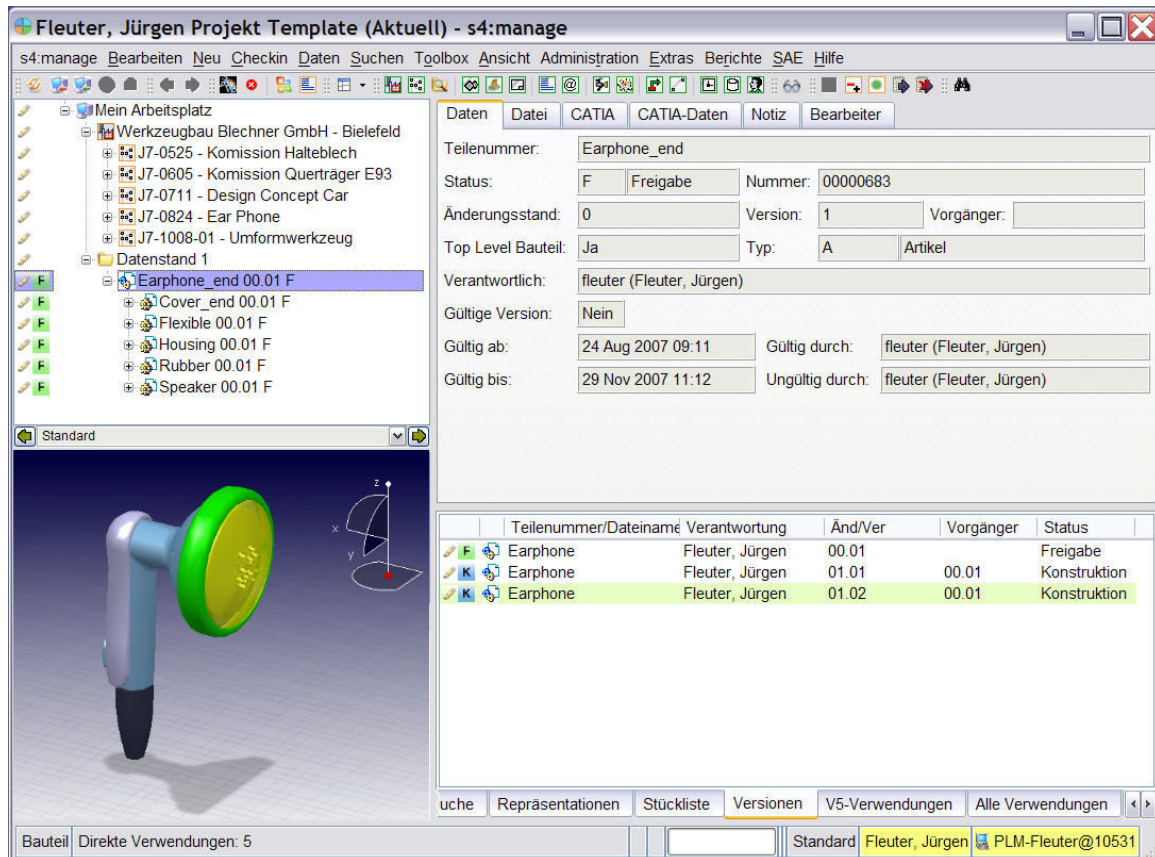


Figure 5. DESYS's s4:manage PyQt Application

German engineering company DESYS (www.desys.de) required a PDM/PLM (Product Data Management/Product Lifecycle Management) application to provide version and release management for document collections, and to provide document format conversions. The solution they created is called s4:manage and it runs on AIX, IRIX, HP-UX, Solaris, Linux, and Windows—and it is written in PyQt. DESYS report: “The combination of Python, Qt and PyQt is one of the most productive solutions for application development, even for very large projects.” They add that: “The ability of PyQt4 to create new widgets in Python and make them accessible in *Qt Designer* via plugins in Python allows us to build libraries of reusable components without the need to implement them in C++”. They also gave away one of the secrets why professional and non-professional software developers alike are so enthusiastic about PyQt: “Building applications with PyQt is just fun.” A screenshot of s4:manage is shown in Figure 5.

One non-commercial PyQt application that is particularly worthy of note is Eric4—an IDE used for program development using Python and many other languages (eric-ide.python-projects.org).

7. Conclusion

PyQt combines Python and Qt into a best-of-breed solution suited for Python developers looking for a GUI framework, and for technical professionals looking for a migration path from Visual Basic. In addition, PyQt is an excellent option for software professionals who need to use agile development practices to rapidly deliver large, scalable, GUI applications.

Since PyQt insulates programmers from platform differences, and from Windows version differences, they can avoid spending unproductive time and energy on dealing with platform and version issues, and focus on actually developing the software that needs to be delivered.

PyQt's wide-ranging capabilities in combination with the ability to easily integrate with existing C and C++ code bases provides both non-software professionals and professional software developers with an agile solution to meet the needs of today's rapidly changing business environments.

If you want to find out more about PyQt, the best way is to download it and try it out. But if you want to read more first, one place to start is PyQt's online documentation. This is very extensive and is supported by many examples that demonstrate the technologies and features that are available to PyQt programmers. For more insight into how PyQt programs are created and how to make best use of PyQt, the book, *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming* by Mark Summerfield (ISBN 0132354187) is the standard PyQt textbook. Another useful source of information is the PyQt mailing list (www.riverbankcomputing.com/pipermail/pyqt), which many PyQt experts contribute to.

Credits

Thanks to David K. Hess who wrote most of the original draft, and to Mark Summerfield for the examples and for the final draft.