Media Engineering and Technology Faculty
German University in Cairo

University of Freiburg
Faculty of Engineering
Institute for Computer Science
Programming Languages

PyDroid: A Python-Based Android Programming Framework

Bachelor Thesis

Author:	Tarek Samir Abd El-Moaty Mostafa Sheasha
Supervisors:	Prof. Dr. Peter Thiemann, University of Freiburg
	Prof. Dr. Slim Abdennadher, German University in Cairo
	Roman Matthias Keil, M. Sc. University of Freiburg
Submission Date:	15 July, 2012

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

---

Tarek Sheasha
15 July, 2012

DECLARATION

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Tarek Sheasha
15 July, 2012
Freiburg im Breigau

Acknowledgments

"Never submit to failure until trying all possible approaches, and trying all possible approaches means success has been reached". This quote has been relentlessly roaming my mind throughout the coarse of this project. Since I first got in contact with Professor Dr. Peter Thiemann requesting his approval of supervision over this project, I have been determined to live up to both of our expectations of accomplishing the required result. However, doing so was not a simple task by any means. I knew achieving such a goal was accompanied by many difficulties. Overcoming these difficulties required motivation, persistence and continuous day in day out efforts. Several people have been by my side during the 3 months I have spent implementing this project and the months before that that lead to me having the opportunity to do this project in the University of Freiburg, 6500 kilometres from my university back in Egypt, the German University in Cairo. I have placed these people into three categories maintaining an equal level of support and appreciation for each category.

Firstly, I would like to offer my deepest gratitude and appreciation to the people that provided me the chance to implement this project in the University of Freiburg. My parents, Dr. Samir Mostafa and Dr. Dalia Beshir, have supported me by all the possible means of support, whether it was educational support from the time I was born, spiritual support that held me in place whenever I reached a matter that would give the thought of giving up and even financial support in every aspect of my life and especially this trip. My professors in Cairo, Dr. Fatma Meawad, Prof. Dr. Slim Abdennadher, Professor Dr. Carmen Gervet and Dr. Haythem Ismail. Without their support throughout my period of study, their recommendations and their devotion towards all the students in general and me in particular to reach the highest levels of success, I would not have made it to this point and reached what I have reached during this project.

Secondly, I would like to give credit to all the technical support and guidance I have received from the respectable personalities I have met during my stay in Freiburg, Germany. Prof. Dr. Peter Thiemann has provided me with endless resources to accomplish my project never hesitating in any decision that could lead to my success of any sort. The Python for Android[11] and PySide for Andorid[13] project developers, Daniel Oppenheim, Anthony Prieur and Thomas Perl respectively have been of ultimate assistance, never leaving any question I have go unanswered and even providing support without my request. I thank you all, you have given me the chance to reach the first goal of PyDroid with you immense support, I could not have done it without you. Last and definitely not least, Roman Matthias Keil, a Ph.D at the University of Freiburg, whose efforts and weekly meetings with me were the backbone of this project. I was amazed with the dedication and devotion you were

showing towards me to accomplish this project, always keeping in mind what is the best for me, even at times when you were extremely busy you never compromised a chance that could lead to even the slightest of success of this project. I thank you once again.

Finally, to all the friends I have met here and the friends I have in Egypt that kept my mental state intact at times when I needed them most. Living away from all the people and places I am acquainted with lead sometimes to depression, homesickness and decrease in productivity. These honourable people however were my major source of inspiration and motivation. Leading these group of people is the recently graduated Applied Arts Major, Nahla El-Gizawy. Simply, you kept me going and you held me together in one piece. I am grateful God gave me the chance of knowing you, and I am forever thankful to Him for that. I thank this very special person once again, you made it all happen.

Abstract

This project is a stepping-stone towards creating a fully functional Python written Software Development Kit (SDK). This SDK could then be use for the Android mobile platform acting as a viable alternative to the current Java written SDK. As a means of reaching this final outcome, two major steps need to be accomplished. First and foremost, a suitable Python GUI Framework was extensively researched to obtain one that perfectly coincides with PyDroid's aim through being able to create widgets with features similar to those from the current Java written SDK. The chosen GUI framework needs then be ported to the Android platform in the form of a Python module. This port will thus allow GUI applications to be written purely in Python code and run on Android's Dalvik Virtual Machine. Secondly, components from the current Android SDK needed to create simple GUI applications on Android devices were implemented in Python. These components are an exact replica of the current widgets used by the Android SDK, written in Java, to create similar applications in Python using a the ported GUI Framework. Applications produced by this Python written development kit could run on Android devices and on a non-Android platforms without the need to use an Android emulator. Furthermore, the created applications have all functionalities provided by similar applications created using the current Android SDK written in the Java programming language. Absolute abstraction comparable to that of the current SDK widgets was maintained in the mimicked Python written widgets, accommodating a wide-range of GUI applications to be developed with no need to re-implement the fundamental widget features. Promising results have been achieved throughout the process of this project. Feasibility to further develop the framework to accommodate a wider range of components that could be used to develop more complex applications was one of those results.

Contents

Chapter 1

Introduction

Controversy has been aroused during the past years between software developers regarding their choice of programming language. This never ending language ethnocentrism is a result of the numerous programming paradigms, languages and techniques. With the rise in number of programmers in the field of mobile application development, this aforementioned controversy has reached its peak. Matter of the fact is there is not a single ultimate programming language, each has its advantages and disadvantages. When it comes to the field of mobile application development, a platform should be available for the very rich developer variety currently present. The reasoning behind this is developers should not be restricted to a certain language and hence a specific paradigm and technique since this might cut down on the productivity and quality of mobile applications.

Thus, it has been a main concern to allow for this restriction to be bypassed. Android application developers are currently using Java as a programming language therefore programming in an Object-Oriented Paradigm. As a developer that uses programming languages, other than Java, primarily when developing applications an obstacle will be faced when creating mobile applications for Android. Consequently, to overcome this obstacle, a frameworks to create Android applications using various programming languages should be made readily available.

1.1 Conceptualizing PyDroid

PyDroid aims at porting a Python GUI Framework to the Android platform and creating a Software Development Kit written in pure Python using the chosen GUI framework. Such a port would allow creation of GUI Applications for Android devices written purely in Python. Python was chosen as the language due to its multi-paradigm characteristics, fully encapsulating the Object-Oriented, functional and logical paradigms. With this advantage over the current single-paradigm development on Android a larger variety of developers would be able to develop on Android, allowing for higher productivity, quality and creativity in the mobile applications. During the coarse of this project the first goal of PyDroid, to port a

Python GUI Framework, is planned to be achieved, as a step toward using this port to create a pure Python written SDK that holds all functionalities of the current Java written SDK.

Python Selection of the language in which the SDK should be developed was concerned with two main issues. Firstly, this language should cover a wide-range of development paradigms. Doing so, will effectively increase developer interest towards creating applications, and reduce the aforementioned language ethnocentrism. Secondly, a language that supports development of the type intended applications and has a solid documentation in that area and support community. Combining these two issues, results in a languages the supports development of GUI applications, possibly through existing GUI tool kits, and completely supports Object-Oriented, functional, logic, and imperative paradigms. The result of this filter was the elimination of several languages, however still the remaining languages were still quite a lot and very hard to choose from. The most prominent languages resulting from this filtration were, Ruby[7] and Python[10].

The decision was resolved by comparing the current development platforms that support both development using these languages. This comparison[1] was available online. Python and Ruby are both available for UNIX, DOS, Windows 95 / 98 / NT / 2000 / Millenium / XP / Vista / 7, Mac OS X, BeOS, Amiga, Acorn Risc OS, and OS/2. Python is also available under AS400, QNX, VMS, OS390, AROS, Windows CE. Given the larger development platform support of Python and hence wider range of developers, Python was the resulting language from this comparison and hence the choice of the language for the SDK.

Android With mobile application development being a rising point of interest for a wide range of developers, it was clear choosing a platform had to satisfy as much of this range as possible. Several market researches have been done in this area to pin point a single mobile platform that satisfies the need of the largest set of these developers. It seemed in each publication the conclusion was, to a certain extent, biassed due to personal experiences of the writer. However, Earl Oliver conducted a research[12], that cohered perfectly to the logic and concept behind this project, by stating a conclusion that there is no single mobile platform that satisfies a large range of developers. It is always debatable, since the choice of a mobile platform to develop is itself based on the application being developed. Thus, a sound choice of the platform for for which this SDK should be developed was taken into consideration. The first point examined was the type of applications that were to be provided for by the SDK. The intended type of developed applications are GUI applications, requiring user interaction, other variables that consist of the programming language choice, the popularity of the platform[2] and possibility to cross-compile language extensions written in *C/C++* to the platform were also a point of interest.

Variable Analysis Python as the language of choice was a very influential factor in choosing Android to be the platform for this project. Google, the owner of

Android, has recently added Guido van Rossum, the creator of Android, to its list of employees. Efforts have since then been focused on using Python as the language for development of several Google products, starting off with Google's web-based code review and storage product [8]. Therefore, it could be concluded that Google's only mobile platform, Android, will soon follow this chain of programming language update.

Since the Android platform was introduced to mobile devices in 2007, its market share has increased drastically[4]. Since a high market share indicates a high user consumption of the product, these users need to be satisfied. To achieve such a satisfaction for the wide range of users, a wide range of developers are needed, constricting these developers however to a single SDK and its accompanying development language will inevitably compromise the quality of the product. Hence, the decision was made to develop the Python written SDK for the Android platform and not any of the other available platforms.

Platform cross-compilation compatibility was a major interest point in the decision process, and Android proved to be very deep and comprehensive regarding this topic. Android development is not done only through the components of the Android SDK, another very important development kit, namely, Native Development Kit (NDK), is crucial when porting language specific modules, extensions and frameworks to the Android platform. Generally, developers tend to use features from several modules of a language while developing an application, not all these modules are readily available on all platforms. Python GUI tool kits have no support for the Android platform, and therefore must be cross-compiled to be usable when developing Android applications. Since GUI tool kits are mostly written in *C/C++* and since the NDK allows for cross-compilation of C/C++ sources to be executable on Android devices, this furthermore solidified the choice of the Android platform.

1.2 Related Work

Android platform developers have been working restlessly in the past years to achieve a multi-lingual development environment. Efforts spent in this area have reached very promising results, some of which are very close in context to the goal of this project. A brief introduction and comparison to the projects with the closest common ground with respect to PyDroid follow.

Scripting Layer for Android (SL4A)[9] This project brings scripting languages to Android by allowing users to edit and execute scripts and interactive interpreters directly on the Android device. Scripts developed using this project have access to several Android APIs used in Android applications, this access can be done with ease due to the projects simplified interface. Running scripts written using this project's interface renders the created applications in HTML pages instead of using Android's Dalvik Virtual Machine to run the scripts as done with applications written in Java using the current SDK. What is provided by this project is more of a library than it is an independent SDK written in each of the scripting languages supported. This

in return causes limitations when trying to create complex applications that need the usage of internal device components. Limitations in that area are not present when programming using the Java written SDK.

PyDroid aims to provide an SDK that produces applications that run on the Dalvik VM and not only a library that renders HTML pages containing the created applications. The Python written SDK aimed for, will be as complete as the current Java written SDK and thus maintain all the currently provided features, not only a limited subset of the Java written SDK.

Python for Android (Py4A)[11] This project is used in conjunction with the previously mentioned project[9], to produce python scripts directly runnable from the Android device Dalvik VM. The developed scripts either use the library from SL4A that is a limited subset of the Java written SDK, or perform background processes that require no interaction with the user in the form of a GUI application.

PyDroid coincides mostly with the ideology of this project to run Python scripts directly via the Dalvik VM, however the area of creating fully functional GUI applications remains untouched by Py4A.

PySide for Android[13] **using Necessitas**[5] Holding the closest common ground with PyDroid, this project fulfills the first goal of PyDroid which is porting a python GUI framework to Android. However, the choice of the GUI framework differs with PyDroid's Python GUI framework. The methodology of determining the Python GUI framework to be utilised by PyDroid will be discussed in the next chapter. Porting a Python GUI framework to Android as previously discussed, is only an initial step to completing the Python written SDK and hence this project alone does not provide a substitute to PyDroid. Another very important difference and the main reason why this project was not enough to fulfill PyDoird's first goal was the use of an old Python version *2.6*, whereas the intended version to use for the Python written SDK is Python *2.7*.

During the coarse of this project, the two latter aforementioned projects have been utilised to achieve the goal of porting a Python GUI framework to Android. Detailed explanation of the components used from each project will be discussed in forthcoming chapters.

Chapter 2

PyDroid Design Principles

Solid reasoning and sound arguments supporting the Python GUI framework of choice are the main aim of this chapter. Obtaining such an aim first specifying what features the framework should be capable of producing, and even replicating to be in absolute coherence with the produced features through Android's current Java written SDK. The main attribute of the sought features are them being fundamental. Fundamentalism of a feature indicates that it is readily available in a stand-alone form within the SDK, not created by conjunction of other widgets. To obtain such a basic widget subset two GUI applications were developed using the current Java written SDK. These applications were a To-Do List Manager application and an RSS Reader. To further emphasize that the extracted Java written SDK widgets are those of the most basic form, the requirements maintained a strictly simple resulting application. Section 2.1 explains in more detail the used methodology to extract this subset effectively.

Following the extraction of the widget subset described earlier, focus should be logically shifted to use the elements of this subset as the standard to decide which framework to use. Python has numerous GUI frameworks and tool-kits that could be ported to Android, however the question lies in which of these completely satisfies all aspects of the widget subset. Accordingly, filtering out the incoherent options and settling for the most suitable substitute takes place. This process of filtration could be divided into three stages. Firstly, the GUI framework must include a declarative GUI interface that coincides with the ideology of the Java written SDK. After applying this filter, and removing only a few options, a further constraint had to be applied. This constraint and hence the second stage of the filtration process targeted those frameworks that had the most extensive documentations available and a dedicated support community comparable to those of the current SDK. Such a constraint is very important since it was also an indication of the most readily used GUI frameworks and tool-kits thus satisfying as much developers as possible. The final stage of this filtration process, after narrowing the options to a minimal number, is to compare the inner details of the framework to those of the current SDK and testing the compatibility of the framework to the Android platform components. Section 2.2 contains the comparison results.

Finally, using the framework of choice abstract classes that act as an interface to the widgets obtained from the previous step 2.1 were implemented. In addition to creating these widget interfaces, they were used to re-implement a Python version of the applications created in the previous section. Accordingly. This mimic of the Java written SDK widgets provided a further compatibility test on whether the framework could be ported to Android. Section 2.3 contains details of mimicking the widget subset into interfaces and creating the applications using these interfaces.

2.1 Fundamental Widget Subset

As discussed previously the task to extract a widget subset constituting of fundamental widgets devised for GUI applications need to be developed. This section includes the requirements that were set for each of these applications to maintain the simplistic level of utilised widgets.

To-Do List Manager This application specifies features used by most GUI applications, regardless of their complexity. Requirements to produce this application were user interaction through a GUI with a method to input and display output and saving, retrieving and editing the program state. This set of requirements ensured the most generic widget subset to be extracted. Following is the set of extracted widgets from this application.

- Input text field to enter the items of the To-Do list.
- Button to add, edit and delete items.
- Database that holds the items of the list.
- View allowing item list to be loaded from the database and displayed to the user.

RSS Reader An RSS Reader has two additional features that were not introduced during the development of the previous To-Do List Manager application, which are *Internet-Access* and *XML Parsing*. Once more ensuring the extracted widgets be of utter fundamentalism both these new features were introduced in their most simple form. No unnecessary features were added to make the application look more beautiful since this is out of the scope of this project's aim. Following is the set of extracted widgets from this application.

- Input text field to enter URL of RSS Feed.
- Internet access to retrieve RSS Feed.
- Button to start receiving RSS Feed.
- Parsing and interpreting XML Feed into items in a list.
- Interactive Dynamic View allowing Feed to be refreshed regularly and displayed to user.

After creating two simple GUI applications, and successfully extracting the subset if fundamental widgets from Android's Java written SDK, deciding on the most suitable Python GUI Framework to implement these widgets is at hand. This framework should have a declarative interface, constitute of object composition and support the extracted widgets. A Python mimic of these SDK widgets will be implemented, by utilizing the GUI Framework modules.

The full list of the extracted widgets from both applications can be found in Appendix A.

2.2 Python GUI Framework Decision

The constraints to choose a GUI framework, previously discussed in the preface of this chapter, were thoroughly studied and applied them to the available Python GUI frameworks in the previous two stages. These first stages of the filtration process resulted in narrowing down the choice to only two frameworks, **PyQt** and **PyGTK**. In this section the third filtration stage is applied. A thorough comparison of internal features of the framework in terms of compatibility with Android components and development requirements was the tool utilised to apply this final filter. Examination of simplicity, functionality and compatibility of the declarative interfaces provided for by each of these frameworks was the main factor in this comparison.

PyGTK PyGTK is a GUI tool-kit based on the *GTK+*[14] tool-kit. *GTK+*, is an object-oriented widget tool-kit created using the C programming language. Running a *GTK+* program is very simple due to its cross-platform nature, nearly embedded in all *Linux* and *Apple* platforms. The latest Python distributions on *Linus* systems include GTK natively built as a part into Python libraries.

The Declarative Interface: *gtk.builder* *PyGTK* comes with a declarative GUI provider via the *gtk.Builder* object. This object reads textual descriptions of a user interface and creates instances of the described objects. The *gtk.Builder* objects can be used to load a file that contains XML data describing widgets for the user interface and callbacks to the events that should be exposed by the code. *Glade* is a tool that could be used to create the XML via a graphical designer interface to avoid writing the XML code manually. Usage of this tool makes debugging of the XML code obsolete. That is due to drawing the application components through a provided interface and generating an XML source rather than creating the XML in a text editor and then running to get a prototype.

PyQt PyQt is a GUI framework, not only a tool-kit, based on Qt[6] which in turn is implemented in C++. *PyQt* contains functionalities that exceed the scope of tool-kits, for example database access. *Qt* provides possibility to perform tweaks and adjustments to specific platforms through a dense cross-platform window service.

The Declarative Interface: *QML* The *QML* language designed for *Qt* is also available when programming in Python Qt binding , *PyQt*. By using such a language user interfaces become more fluid, in terms of object composition, in the sense that simple elements are built up to make components. States, such as *onFocus*, are also available to be applied on objects within this language. *QML* provides building blocks for creating widgets that could be readily used and modified to suit a wide range of application needs. A *QObject* is an instance of an element of the *QML* language existing in the *PyQt* framework. This allows manipulation of the object in a declarative way as mentioned earlier before by setting values and properties in a respective sense without putting into consideration how that is done. By extending *QObject* instances or any of its subclasses into the code it is possible to connect objects to event handlers via *Slots* and emit events as *Signals*.

Integrating Python with the *QML* language is simple. However, three types of *QML* applications are available not all are possible when developing in Python. *QML* code for the applications can be generated through drawing the objects using a tool called **QtCreator**.

QML Application Types

- Pure *QML* Applications with no Python back-end, this is fully supported by *PyQt*.
- Using *QObject* instances in a Python back-end of a *QML* application. This also is fully supported by *PyQt*.
- Using *QObject* sub-classes in *QML* applications as to create new instances from within *QML* code. However this is not supported by *PyQt* since *QML* uses information generated statically at compile time rather than information created dynamically at run time. This approach works fine for languages such as *C++* however they do not function with the dynamic nature of Python. However, *QML* also allows *QObject* sub-classes to be used in *QML* applications so that new instances are created from within *QML* code. Such a restriction does not affect the functionality of Python-QML applications rather the restriction is placed on how the application is written.

Comparison of both frameworks with respect to features in previous introduction to both and other intra framework features is the final stage of this filtration process.

Android Compatibility *PyQt* uses the SQLite3 database which is the one used on Android devices, *PyGTK*'s database is of a different architecture and not supported by Android Devices.

Declarative Interface *PyGTK* has the declarative interface in the form of the *gtk.Builder* object and accompanying XML code, that unlike *PyQt*'s well established *QML* to *QObject* integration does not natively provide for changes in state of an object, for example *onFocus*. *PyGTK*'s declarative interface tools can not be used to create standalone applications unlike *PyQt* where standalone *QML* applications can be created.

Design Tools *PyQt* uses QtCreator to design QML which includes a seem-less designer and functions like an IDE with an extension to deploy *C++ Qt* applications to Android. Glade, used by *PyGTK*, is only a designing kit with no ability to run or deploy applications created using it.

Documentation Both frameworks have a very extensive documentation and a well founded support community.

Memory Usage *PyGTK* uses slightly less memory than *PyQT* on a given device, however given a new C compiler they both have similar memory usage rates.

Portability *PyQt* has distributions for all platforms and an established window event system that is platform independent, whereas *PyGTK* is only available for the major platforms and uses a very platform dependent window and event system.

Visual Output *PyQt* looks native and provides consistency when moving across platforms. *PyGTK* looks less native yet also provides similar cross-platform consistency to *PyQt*.

Conclusion After an extensive filtration process, only one of the frameworks only should pass the final filter and thus ported to Android. Given the advantages and disadvantages of each framework previously discussed, and relativity of each framework to the goal aimed for by PyDroid, the chosen framework to use in this project is ***PyQt***. To sum up, *PyQt* has shown features such as compatibility with Android internal components, a profound declarative interface, a portable window and event system and a consistent visual output across all platforms. These features provide the leverage needed for *PyQt* over *PyGTK* to be used as the Python GUI framework for PyDroid.

2.3 Mimicking the Widgets

In this Section, the Android SDK widgets extracted earlier in Section 2.1 will be mimicked in Python using modules from the *PyQt*. Mimicking the SDK widgets does not only aim to provide the same functionality of the Java written SDK widgets, however the same final visual output should also be put into consideration. The reasoning behind mimicking the widgets lies in proving or disproving the concept of creating an alternate Android SDK written in Python. If the mimicked widgets level out all the features and functionalities of the Java written SDK widgets, then the only logical next step after porting the GUI framework to Android is to complete this alternate SDK with all the other available widgets of the current Android SDK.

Database access widgets utilised by To-Do List application and internet access widget and XML parsing widgets used by the RSS Reader application need not be mimicked since they are readily available parts of the Python language libraries.

These readily available libraries can simply be imported and integrated within GUI widgets that will be mimicked. Therefore placing the advantage of this SDK, at a very early stage, of being a light-weight SDK with new classes and interfaces being created solely for GUI purposes, using *PyQt*, and not internal application functionalities. This advantage is valid since the current Android SDK contains specific widgets and packages that deal with functionalities that are not available in the Java libraries, bur are however readily available in Python libraries.

Mimicked Widgets The mimicked widgets include built-in connection and access to an SQLite3 Database via Python libraries, with the ability to add, edit and delete items on a list. Additionally, buttons along with their listeners and action events are also built into the mimicked widget templates.

1. ListView widget with built-in ListAdapter widget that provides database access.
2. Listitem widget that contains details of an item on the list and buttons to edit or delete the item.
3. Button widget with built-in listener and action event (signals & slots).

The mimicking process consists of two parts, QML definition (declarative GUI) and a Python back-end to interact with the GUI components. During this process it was ensured to maintain ultimate abstraction in the mimicked widgets. This is to provide a flexible interface for developers for the templates to fit any simple GUI application without the need to re-implement any of the widgets. The use of Python as the language for development has made it very simple to maintain this high level of abstraction given Python's dynamic object creation feature. In other words any objects within these mimicked widgets abstracted interfaces can be customised from within the developers application code, whether the customisation is to add, remove, edit, or even inspect all the available features fields of the widget. These templates provide the most simplistic form of the SDK widgets, yet feasibility to create the more complicated form of the widget exists through the conjunction of these widgets. However, limitations exist since the mimicked widgets are only a small subset of the SDK widgets and hence at this stage it is not possible to create the most complex GUI applications. This limitation is only due to the fact that the Python written SDK is not close to being complete at this stage. Hence it is out of the scope of this project, which aims at providing the GUI framework port to develop the Python written SDK at a later stage, to overcome this limitation.

Execution Environments Applications created using these widgets could be directly executed and run on a regular computer or on an Android mobile device, without addition or removal of code to cope for this platform change. With such an option at hand, of where to run the application, three very important advantages arise. Firstly, Android applications will be readily available on different platforms with the same exact features, functionality and look on all these platforms. Secondly, created applications could be tested on any platform without the need to

use an Android Emulator to run the application first. Finally, no matter on which platform the application has been developed and tested it is guaranteed that the application will run exactly in the same manner on an Android device or any other platform once again without editing the application code by any means.

Appendix B contains the source code of one of the mimicked widgets, namely the ListView widget. A To-Do List Manager application was created using the mimicked widgets and executed on a Linux-x86 Machine. Also included in Appendix B is the source code used to create the previously mentioned application and a snapshot of its running.

Note: The code density to create such an application is noticeably less than the alternate Java implementation using the current Android SDK widgets.

Chapter 3

Porting to Android

With a chosen Python GUI framework to be the tool to write out the alternate Python written Android SDK, focus is now shifted on how to make this framework available on Android. Two approaches were proposed, yet only one of these was the most compatible to the logic behind PyDroid. The following section contains a more detailed explanation concerning these two approaches.

3.1 Porting Proposals

The first approach constituted of writing the application source code in Python and translating the Python code to Java code accordingly using the current Android Java SDK. The second, more appropriate, approach stated that written applications in Python should be run directly as is without modification on Android devices.

3.1.1 First Approach: Python to Java

Efforts have been spent in the recent years to obtain such a language translation. However, all those efforts, mos prominently Jython[3], have no support for Python's GUI framework terminology. These efforts aimed at transforming Python scripts strictly in the field of non-GUI development to Java bytecode. Such a restriction made it obvious this technique will be in appropriate to fulfill this approach. Another technique was considered, this new technique was more realisable and plausible. Python application code will be transformed to Java code utilising Android SDK widgets through parsing the source, using Python's *Abstract Syntax Tree* class. Once the code is parsed and tokens for each keyword were produced the code is to be translated to the corresponding Java code with SDK widgets. A quick glimpse over this new methodology does not seem to hold any problem, yet after a deeper analysis impracticality of this technique aroused. This impracticality lies in the dynamic nature of Python and hence that of the GUI framework, in comparison to the compiled Java language. Trying to parse a GUI application created using *PyQt* alone is itself impractical, since object referencing and attribute assignments are done dynamically within the code, thus impossible to track which object belongs

where and with which attributes just by parsing the code. Finally, re-stating the aim of PyDroid made it clear that this approach as a whole is not the sought approach. PyDroid, as previously stated, aims to provide an alternate Android SDK written in Python to allow a wider-range of developers to start developing Android applications. This aim of PyDroid does not coincide with translating Python code to Java code however it is quite contrary to it.

3.1.2 Second Approach: Running on Android's Dalvik VM

Perfectly cohering with the aim of PyDroid, this approach proposes the integration of the Python interpreter and the *PyQt* framework to the Android platform. Such an integration would support running Python-based GUI applications directly on Android's Dalvik VM, with the ability of utilising the device components if and when necessary. This integration requires cross-compiling the Python *2.7* interpreter as well as the *PyQt* GUI framework. Cross-compilation is necessary since neither of the aforementioned necessary components have an Android distribution. Necessity of cross-compiling is due to Androids processor architecture *ARM* that is not supported readily in Python nor *PyQt* distributions. To obtain this cross-compilation two projects in particular have been used and components from each have been integrated with each other to achieve the sought result. These two projects have been discussed in Chapter 1, namely Python for Android[9], and the Andoird Qt Port Necessitas[5] used by the PySide for Android Project[13].

The decision was taken to take the latter approach as the one for implementing the first goal of PyDroid. The next task, and main task of this project is to effectively cross compile all aspects of the Python interpreter and *PyQt* framework and integrating both. The result is a complete and sound development environment for the Android platform with Python as the programming language and *PyQt* as the GUI framework and Python-Qt binding.

3.2 Cross-Compilation Procedure

Before indulging in the process of cross-compiling, the target platform for which this cross-compilation need be done was thoroughly examined. Examination of the target platform places constraints on how the compilation process should proceed through defining valid *C/C++* compiler flags and options. The tool used for cross-compilation on Android is the Native Development Kit (NDK). The Android NDK contains *C/C++* compilers that produce output compatible to Android's *ARM* architecture. A very important notice is, any cross compiled library has to be in the form of a shared library and then loaded from within the Android device on demand. At this stage the NDK was further examined and cross-compiled tools for loading and linking shared libraries were also present. These tools also allow for testing the output files on any platform to check compatibility to the Android platform. With all the tools to perform the cross-compilation at hand, specific compiler flags and options had to be set for both the Python *2.7* interpreter and the *PyQt* framework. Compiler flags and options are set within each library's *Makefile*. A *Makefile* contains the list of C/C++ source files to be compiled, and the very crucial section of linking the output files into a single shared library *.so file* to be loaded from within the Android device. Each of Python *2.7* and *PyQt*, have specific compiler flags that need to be set to enable portability to Android.

Cross-Compilation was performed twice for each to be compatible with both Android's CPU *ARM* architectures, *armeabi* and *armeabi-v7a*. Both runs for the

different architectures were performed on a host platform device, *Linux-x86 machine running Ubuntu 12.04*. The version of the Android NDK used was *NDK r6b*. Compilers used for *C/C++* compilation were of version *4.4.3*.

Python *2.7* Bash scripts and patches were used to generate the *Makefiles*. These scripts and patches were available from the project Py4A[11], however they were modified slightly for different architecture compilation. The *C/C++* sources of the Python language were compiled against Android platform specifications to produce a shared library *libpython2.7.so*. The Android specific compiler flags that were needed for compilation and linking of the Python source files are provided an infrastructure to determine for the flags that need to be added to a Python GUI framework with *C/C++* source files.

C/C++ compiler flags

-m Specifies the machine CPU architecture for which the C source files are built against. The options for this flag were *-mandroid*, *-march=armv5te*, *-marm*, *-msoft-float*, *-mthumb* and *-mtune=xscale*.

-O Specifies the compiler optimization level. For android the optimization level was set to *-O2*.

-W Specifies the types of warning the compiler should display and hence abort compilation if existent. The used options for this flag were, *-Wall*, *-Wstrict-prototypes*, *-Wl,–no-undefined*, *-Wno-psabi*.

-f Specifies compiler features when treating extern inline functions in the code. The used options for this flag were, *-fomit-frame-pointer*, *-fprofile-generate*, *-fprofile-use*, *-fprofile-arcs*, *-ftest-coverage*, *-fno-strict-aliasing*, *-finline-limit=64*, *-fPIC* and *-fsn*.

–sysroot Specifies the root directory where the compiler source files are located, this was used since the default root directory is of the host machine's compiler sources, and since a different compiler is being used explicit definition of the source root directory is needed.

-D Specifies definitions of parameters to be used in source files in the form of the *#define* function. This flag is very useful in situations where source files have several implementations of a certain function for several architectures, and hence this definition specifies which implementation the compiler should process to the output file. Options used for this flag are numerous and differ between modules, however some options are common for all modules. The common options used were, *-DNO_MALLINFO1*, *-D__ARM__ARCH__5*, *-DNDEBUG*, *-DPLATFORM_linux3* and *-DPy_BUIL_CORE*.

-I Specifies path of header files for the source files within a module. Header files of the compiler source are also added since as previously mentioned the compiler is different from the host machine's default compiler.

-o A boolean flag specifying that the output files from the compilation should be in the form of a binary *.o* file and not the usual common binary executable *.out* file.

-c Another boolean flag specifying that the output file should not be linked to any other output file at this stage, yet the linking will be moved to another stage of compilation.

Linker flags

-shared Specifies that the output of linking all the *.o* files should be a shared library *.so* file.

-l Specifies static or shared libraries that should be linked to the resulting shared library after linking. The libraries specified by this flag will be loaded each time the resulting shared library from linking is referenced. The options set for this flag were *-lgcc*, *-lm* and *-lpython2.7*. Another very important library was needed, *libpthread.so*. This library contains functions for process thread functionalities referenced from within source files. However, unlike when the host machine's default compiler, Android's compiler provided by the NDK has this library natively built into its sources and hence need not be referenced as a separate library to be loaded.

The result of the cross-compilation was a python shared library that could run standalone on Android devices without needing to use any Java back-end to do so. The *Python Interactive Interpreter* is also available for usage via terminal using the *adb shell* command. Before running this interpreter successfully environment variables have to be correctly set.

PYTHONHOME=location of root directory Python library and binary sources on device. PYTHONPATH=location of Python library sources and modules. LD_LIBRARY_PATH=location of all linked shared libraries from the linking process describes earlier.

PyQt Before *PyQt* can be successfully cross-compiled, a tool for binding Python to C/C++ needs first to be cross-compiled. Such a tool is needed since *Qt* is written in the C and C++ languages. The tool used by *PyQt* is called *SIP*. This tool runs on *.sip* files included in the *PyQt* framework libraries to generate the C and C++ code for all the framework's modules with bindings to the Python language. The majority of the flags have been described in the earlier section, however the difference lies in the options chosen for each flag. Please refer to the previous section Python *2.7*, for details on the function of each flag.

SIP: C/C++ compiler flags

-f The options used for this flag were *-fomit-frame-pointer*, *-fno-strict-aliasing*, *-finline-limit=64* and *-fPIC*.

-m The options used for this flag were *-mandroid*, *-march=armv5te* and *-mthumb*.

-O The optimization level was set to *-Os* enabling all level 2 optimizations that do not increase code size.

-W The options used for this flag were *-Wall*, *-Wno-psabi*, *-Wl,-z,noexecstack* and *-Wl,-rpath=PATH_TO_RELATED_LIBRARIES*.

–sysroot Points to compiler source directory.

-I Include path to all header files of the SIP module and the compiler.

-D The options set for this flag were *-DNDEBUG* and *-D_REENTRANT*.

-o and *-c* were also used.

SIP: Linker flags

-L Specifies directory of shared libraries that will be linked against this module.

-l The libraries that were to be linked to were *-lstdc++*, *-lsupc++*, *-llog*, *-lz*, *-lm*, *-ldl*, *-lc* and *-lgcc*.

-shared flag was also used.

The output shared library was added the the PYTHONPATH and LD_LIBRARY_PATH environment variables to be registered as a Python module. PyQt is now ready to be cross-compiled and ported to Android as a Python module.

The *PyQt* framework has several internal modules, however not all of these modules were meant to be ported to mobile devices. The *QtDesigner* module provides classes that allow developers to create custom widget plug-ins for Qt Designer, a tool only available on development machines. Hence this component was not added to the cross-compilation process. An issue that is of utter importance is the choice of *PyQt*'s window and event system. There are currently available systems are *X11* for the Linux platform, *WIN* for Windows platform, *QPA* for the Apple Mac platform and *QWS* that works for embedded Linux platforms. The *QWS* window system was thus the chosen window and event system to use while compiling the C and C++ sources of *PyQt*.

Generation of *Makefiles* for *PyQt* is done through the *qmake* tool. This tool is available on the host device platform, however could not be used for the purpose of being specific to create specifications within the *Makefiles* for the host platform. The option to manually create the *Makefiles* was no option since it held a huge risk of error and would have been very time-consuming. A version of the *qmake* tool, compatible with embedded Linux devices, had to be created. The Necessitas project[5], included cross-platform binaries for the *qmake* tool and hence this version was used in order to generate the *PyQt Makefiles* for Android devices. Not only, does Necessitas[5] provide cross-platform binaries that could be used for Android devices specifically, included in the Qt port are cross-compiled *Qt4* shared libraries for Android that need to be dynamically linked respectively to *PyQt*'s modules. An

intelligent deployment service that was created by the Necessitas project[5] creators, named Ministro, serves to extract the compatible *Qt4* libraries to the Android device at the time of deployment. This service runs only once on the first run of the GUI application created using the *Qt* framework or any of the frameworks language-bound frameworks. With all the necessary tools to start cross-compilation in reach, *PyQt* can now be compiled to run on the Android platform, and achieve the first goal of PyDroid.

PyQt: C/C++ compiler flags

-f The options set for this flag were, *-fPIC*, *-fomit-frame-pointer*, *-fno-strict-aliasing* and *-finline-limit=64*.

-m The options used for this flag were *-mandroid*, *-march=armv5te* and *-mthumb*.

-O The optimization level was set to *-Os* enabling all level 2 optimizations that do not increase code size.

-W The options used for this flag were *-Wall*, *-Wno-psabi*, *-Wl,-z,noexecstack* and *-Wl,-rpath=PATH_TO_RELATED_LIBRARIES*.

–sysroot Points to compiler source directory.

-I Include path to all header files of the respective PyQt module from Necessitas's Qt4 for embedded Linux and compiler header files.

-D The options set for this flag were *-DNDEBUG*, *-D_REENTRANT, -DQT_NO_QWS_TRANSFORMED -DQ_WS_QWS* and *-DQT_NO_DEBUG*. Other options were set for each module respectively defining the module's name.

-o and *-c* were also used.

PyQt: Linker flags

-L Specifies directory of shared libraries that will be linked against this module.

-l The libraries that were to be linked to were *-lstdc++*, *-lsupc++*, *-llog*, *-lz*, *-lm*, *-ldl*, *-lc* and *-lgcc*. Libraries from the Necessitas project[5] *Qt4* for embedded Linux were also added to each module respectively.

-shared flag was also used.

Testing Testing of the port was done after using the Android NDK *strip* tool to remove all compilation symbols of the host machine. The testing process was conducted on the produced shared libraries, *.so files*, in two simple yet effective stages. Firstly, using the NDK tool *objdump* with the flag *-x* to check the contents of the *ELF* files of the libraries while linking to the *Binary File Descriptor* (BDF) library. This test will show any compilation errors of the shared libraries, however none were found as a result of this test. The second deeper phase of testing was conducted using NDK's *readelf* tool. Testing using this tool is done without linkage to the *BFD*, and hence any internal inconsistencies will not pass silently. The result of this test returned negative with respect to compilation errors.

Finally, the testing of the port is complete and the NDK shared library loader, *ld*, tool was used to load the libraries to check for any undefined object references. An initial run of this tool on the shared libraries produces successful results on all except for the library corresponding to *PyQt's QtGui* module. Tee result of loading the aforementioned module's shared library was a *Segmentation Fault*. The reason behind this was that the QtGui module is in itself dense and uses up most of the memory stack specified for the application, yet the *SIP module* and the Python *2.7* shared libraries were loaded every time this module was referenced. Loading those extra libraries required space that is not available since the *QtGui* module has already used up most of it. Looking into implementation of *PyQt*'s modules source files hinted out that dynamically linking shared the previously libraries is redundant. The redundancy is due to the fact that whenever a *PyQt* module requires a specific feature from any of *SIP* or Python *2.7* modules only this feature was loaded form the respective shared library using a very efficient algorithm. However the constraint that will make this work is adding the directory location of the *SIP* and Python *2.7* shared libraries to the environment variable PATH. Removing this unnecessary library dependency and correctly setting the PATH environment variable, lead to the *ld* tool being able to successfully load the libraries with no undefined object references or segmentation faults.

Realisation of this *Segmentation Fault* error and hence realisation of this intelligent loading technique of *PyQt* provides a strong support point for *PyQt* as compared to another Python to Qt binding such as *PySide*. *PySide* links all modules to its respective code generator comparable to *SIP*, named *Shiboken*, and also links all modules to the Python shared library.

Chapter 4

Conclusion

PyDroid has achieved its first goal throughout the work of this project, gaining leverage to begin in the next and final phase of creating a sound and complete Python written Android SDK. GUI Applications written in Python are runnable on Android devices and non-Android machines on all platforms given the ported *PyQt* GUI framework to the Python language. Achieving this result was done through a series of phases. Firstly, extensive comparison of available Python GUI frameworks under the *GPL* License terms. Terms of this comparison focused clearly on ease of development and well established support communities, compatibility with Android platform components and most importantly equal level resulting applications directly comparable to those produced by the current Java written Android SDK. Secondly, efforts were concerned with creating the first set of mimicked widgets written in Python needed to produce simple GUI applications. At this phase the aim was to make a feasibility analysis of PyDroid, to check plausibility of using the chosen framework from the first phase to produce GUI applications with comparable quality to applications created using the current Android SDK. Achieving this feasibility constituted of first creating the GUI applications using Android's current SDK, and then re-implementing the applications with the mimicked widgets. Once both versions of the application were complete internal features produced of the first application were successfully mimicked in the latter application. Only then did the final stage of this project immense after fulfilling the feasibility constraint. In this final stage, the chosen GUI framework from the first phase was cross-compiled on a host machine to be runnable on Android. The GUI application created in the second phase of this project was then deployed to an Android device. Success was at hand after the application proved to run while perfectly utilising the needed Android device components. The visual output of the application however is not of the top-notch quality but this feature is out of the scope of this project, yet should definitely be improved in the future during the next stage of PyDroid, where the full set of widgets provided for by the Android's current Java written SDK are to be replicated using the Python language and the *PyQt* GUI framework.

Furthermore, to be able to deploy Python GUI applications on Android devices the Dalvik Virtual Machine, the cross-compiled Python library with the ported *PyQt* module should be available on the device. An initial thought was to manually add the cross-compiled libraries to the device once and this would be sufficient for running

the Python scripts. This procedure was impractical since this required the devices to be rooted. Since, not all Android users are willing to, or acquire a rooted device another solution had to be sought. A further thought was to include the library and all needed dynamically linked shared libraries in the resource directory of any created Android Application package (APK) of a Python-based GUI application. However, this gives the disadvantage of a large *APK* file size compared the the size of *APK* files of Java written GUI applications. To overcome this disadvantage and once more reimbursing the feasibility of PyDroid, a PyDroid APK was devised based on the Python for Android[11] project application and the Pyside for Android[13] project application. This application extracts the Python library created during the coarse of this project along with the required dynamically linked shared libraries for the *PyQt* framework, to the device and updates the environment variables accordingly to allow the extracted Python library to be visible device-wide by all applications. However, there exists a limitation at this stage, to deploy a Python-based GUI application the use of the Scripting Layer for Android[9] library to execute Python scripts on the Android device. The created PyDroid APK accommodates for this limitation, yet not cancelling it out. The PyDroid APK handles the script execution procedures via the aforementioned technique. Python scripts used to create the GUI application are placed in a compressed folder and added to the PyDroid APK resource directory. The scripts are then extracted and executed. Possibility to bypass this limitation will emerge when the final phase of PyDroid is complete, after this phase the created Python written Android SDK will handle the execution of the scripts and allow the creation of application APKs with source files written only in Python.

Finally, this first stage of PyDroid has been successfully complete and preparing for the next stage is at hand. A few features of the GUI applications have not been tested since on a small scale these tests are not possible. The most prominent of these tests is the application *runtime* compared to an equivalent application written in Java. To be able to effectively test this feature more complex GUI applications, that are not currently in the scope of this project, are required. However, with the first goal achieved it is possible to compare the outcome of this project to that of other projects discuss in the Related Work section earlier in Chapter 1. As expected PyDroid was able to create GUI applications written in Python and not only run Python scripts in the backend or by rendering to HTML pages like, Py4A[11] or SL4A[9] projects respectively. The output of the PySide for Android Project[13] is very close to the outcome of PyDroid however the only difference lies in the deployment of the Python libraries. PySdie for Android[13] requires the device to be rooted whereas PyDroid requires no such prerequisite. With all the positive and negative results achieved the next stage of this project should now be feasible to commence while capitalising on the positives and overcoming the negatives.

List of Figures

Appendix

Appendix A

This chapter contains the full set of widgets extracted when creating two simple GUI applications using Android's Java SDK. The first section includes all the widgets extracted from the To-Do List application and the functionality of each widget. The second section specifies the additional widgets needed to create the RSS Reader application using the Java SDK. Widgets listed in the first section were also used while creating the RSS Reader application in the second section, however not listed as to avoid redundancy.

A.1 To-Do List Manager

AndroinManifest.xml A mandatory file in the root directory of all Android projects which provides the Android System with all the essential application information for the code to be runnable on the machine. This information includes :

- Name of Java package of Application.
- The composition of the 4 aforementioned components of an Android Application as per the current application's requirements. Without this the Android System will not know the components and their launching conditions.
- Application host components and Application permissions to access system and API resources and permissions for other applications to interact with local Application components.
- Declares minimum API level for application to run and libraries to be linked to.

strings.xml and drawable directory An XML file in which all strings the application uses must be defined in order to be used in *AndroinManifest.xml*, also used for internationalization of application. The drawable directory is where all images will be placed for the System to access also via *AndroinManifest.xml*.

R.java This file keeps up with the *AndroinManifest.xml* file in a manner that helps the programmer call functions on it for example, R.layout, R.strings to enable an easy interface when writing down the application's source code.

android.content Out of this package 3 components were used, following are their specifications:

- ***content.Intent:*** An Intent is the means by which a program can interact with other programs. It can start new Activities, send to Broadcast Receivers and communicate with background Services.
- ***content.ContentValues:*** Stores a set of of values that can be processed by a ContentResolvers.
- ***content.Context:*** Allows access to Application-specific resources and classes as well as up-calls for application-level operations such as launching activities, broadcasting and receiving Intents.

android.database Out of this package 4 components were used, following are their specifications:

- ***database.Cursor:*** Given a result set returned by a database query, performing random read-write access to this set is provided by this Interface.
- ***database.SQLException:*** SQL execution or parse errors detection is done here.
- ***database.SQLiteDatabase:*** Exposes methods to manage Database actions.
- ***database.SQLiteOpenHelper:*** Manages Database creation and version management.

android.os A mandatory file in the root directory of all Android projects which provides the Android System with all the essential application information for the code to be runnable on the machine. This information includes:

- ***os.Bundle:*** Used to hold data for Intents in general by mapping strings to parcelable data.

android.view Out of this package 3 components were used, following are their specifications:

- ***view.Menu:*** Manages items in a menu, in this application used to make it applicable to create and edit notes via Android System Menu.
- ***view.ContextMenu:*** Provides functionality to edit content of the context menu.
- ***view.MenuItem:*** Editting and Managing Items on a menu.

android.widget Out of this package 6 components were used, following are their specifications:

- ***widget.AdapterView:*** The superclass for the below ListView allows different layouts of data items.
- ***widget.ListView:*** Shows items in a vertical scrollable list.
- ***widget.SimpleCursorAdapter:*** An easy adapter to map columns from a cursor to TextViews or ImageViews defined in an XML file.
- ***widget.AdapterContextMenuInfo:*** Extra menu information provided when a new ContextMenu is brought up for this AdapterView.
- ***widget.Button:*** A push-button widget, that also provide listeners for the button through which actions events are handled and processed.
- ***widget.EditText:*** EditText is a thin veneer over TextView that configures itself to be editable.

A.2 RSS Reader

For this Application to be fully functional more classes were needed from the *sxaproject* library, the *Java* library and the *Javax* library. These classes are out of the scope of this study, which are widgets used for GUI application development, and hence were overlooked.

android.widget Out of this package 2 components were used, following are their specifications:

- ***widget.AdapterView.OnItemClickListener:*** Handles Callback event once an item in the Adapter view is clicked, basically an Event Handler.
- ***widget.ArrayAdapter:*** This class fills in TextView elements sent via constructor of ArrayAdapter Class.

Appendix B

The source code of the created abstract class to be used as an interface when creating the *ListView* widget in the Python SDK using *PyQt* is presented in this chapter. Such an example provides a comparison to the Java SDK class of the same widget with respect to code density and functionality.

B.1 Python Mimic of *ListView* Widget

Each mimicked widget contains two parts a *QML* Figure B.1 GUI section and a Python Figure B.2 script backend. This is yet another advantage of *PyQt* since it is able to separate the GUI code from the Python code very effectively.

B.2 Utilizing the Mimicked Widgets: PyQt To-Do List Application

As a further tool for comparison to the Java SDK, the following code snippets show how the created interfaces can be imported into the code of a GUI application and tweaked to suit application-specific requirements. The snippets include the GUI *QML* Figure B.3 code and the accompanying Python Figure B.4 script backend. The resulting application Figure B.5 running on the development machine is also included.

Figure B.1: QML source for the ListView widget.

```
1  import Qt 4.7
2
3  ListView {
4          width: parent.width
5          height: parent.height*0.8
6          boundsBehavior: Flickable.StopAtBounds
7          anchors.horizontalCenter: parent.horizontalCenter
8          anchors.verticalCenter: parent.verticalCenter
9
10         model: items
11
12         delegate:
13         NListItem {
14             opacity: 0.7
15             anchors.leftMargin: 0
16         }
17     }
```

Figure B.2: The Python source for the ListView widget.

```
1  from PyQt4 import QtCore, QtGui, QtDeclarative
2  from PyQt4.QtSql import *
3  from list_item import *
4  class ListView(QtCore.QObject):
5      itemsChanged = QtCore.pyqtSignal()
6      def __init__(self, new_rc, connection):
7          super(ListView, self).__init__()
8          self._items = []
9          self._rc = new_rc
10         self._con = connection
11         with self._con:
12             self._cur = self._con.cursor()
13             self._cur.execute("CREATE␣TABLE␣IF␣NOT␣EXISTS␣Items
                   (Item␣TEXT)")
14         self._cur.execute("SELECT␣*␣from␣Items")
15         row = self._cur.fetchall()
16         for r in row:
17             t = TodoItem()
18             t.set_text(QtCore.QString(r[0]))
19             self._items.insert(0,t)
20     @QtCore.pyqtProperty(QtDeclarative.
          QPyDeclarativeListProperty, notify=itemsChanged)
21     def items(self):
22         return QtDeclarative.QPyDeclarativeListProperty(self,
              self._items)
23     @QtCore.pyqtSlot(QtCore.QString)
24     def add_item(self, txt):
25         self._cur.execute("INSERT␣INTO␣Items␣VALUES(?)", [str(
              txt)])
26         self._con.commit()
27         t = TodoItem()
28         t.set_text(QtCore.QString(txt))
29         self._items.insert(0, t)
30         self._rc.setContextProperty("items", self._items)
31         self.itemsChanged.emit()
32     @QtCore.pyqtSlot(int)
33     def remove_item(self, to_remove):
34         self._cur.execute("DELETE␣FROM␣Items␣WHERE␣Item␣=␣?", [
              str(self._items[to_remove]._item_text)])
35         self._con.commit()
36         self._items.pop(to_remove)
37         self._rc.setContextProperty("items", self._items)
38         self.itemsChanged.emit()
39     @QtCore.pyqtSlot(int, QtCore.QString)
40     def edit_item(self, to_edit, text):
41         self._items[to_edit].set_text(text)
42         self._rc.setContextProperty("items", self._items)
43         self.itemsChanged.emit()
```

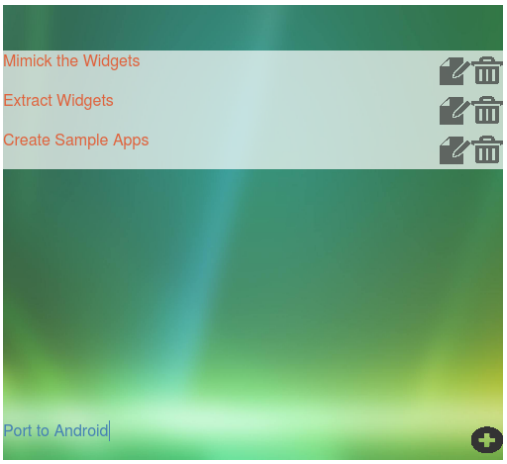
Figure B.3: QML source for the To-Do List Manager Application

```
import Qt 4.7

Rectangle {
    transformOrigin: Item.Center
    smooth: true

    Image{
    width: parent.width
    height: parent.height
    source: "background.jpg"

    NListView{}

    NTextInput {
            id: input_text
     MouseArea {
            width: 32
            height: 40
            anchors.right: parent.right
            anchors.verticalCenter: parent.verticalCenter
            anchors.rightMargin: 0
        NButton{
            anchors.fill:parent
            source:"add.svg"
        }
         onClicked: {
            add_item(input_text.text), input_text.text=qsTr("")
         }
     }

    }
  }
}
```

Figure B.4: Python source for the To-Do List Manager Application

```
from PyQt4 import QtCore, QtGui, QtDeclarative
from list_view import *

if __name__ == '__main__':

    import sys
    import sqlite3 as lite

    app = QtGui.QApplication(sys.argv)
    con = lite.connect('mydb.db')

    canvas = QtDeclarative.QDeclarativeView()
    rc = canvas.rootContext()

    todo = ListView(rc, con)

    rc.setContextObject(todo)
    canvas.setSource(QtCore.QUrl.fromLocalFile('todo.qml'))
    canvas.setGeometry(QtCore.QRect(100, 100, 350, 350))
    canvas.setResizeMode(QtDeclarative.QDeclarativeView.
       SizeRootObjectToView)


    canvas.show()
    sys.exit(app.exec_())
```

Figure B.5: To-Do List Manager Application



Bibliography

[1] Python Vs. Ruby. http://c2.com/cgi/wiki?PythonVsRuby.

[2] Computer science and network technology (iccsnt), 2011 international conference on. volume 4, pages i –ii, dec. 2011.

[3] R.W. Bill. *Jython for Java Programmers*. Landmark Series. New Riders, 2002.

[4] Gadhavi Bimal. Analysis of the emerging android market.

[5] BodDan Vatra. Necessitas: A Qt Port for Android. http://sourceforge.net/p/necessitas/home/necessitas/.

[6] M.K. Dalheimer. *Programming With Qt*. O'Reilly Series. O'Reilly, 2002.

[7] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly, 2008.

[8] James Gray. Interview with guido van rossum. *Linux J.*, 2008(174), October 2008.

[9] Lucas Jordan, Pieter Greyling, Lucas Jordan, and Pieter Greyling. *Introducing SL4A: The Scripting Layer for Android*. Apress, 2011.

[10] Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[11] Matthews, R. Python for Android Project. http://code.google.com/p/python-for-android/.

[12] Earl Oliver. A survey of platforms for mobile networks research. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(4):56–63, February 2009.

[13] Perl, T. PySide for Android Project. http://thp.io/2011/pyside-android/.

[14] Peter Wright. *Beginning GTK+/Gnome Programming*. Wrox Press Ltd., Birmingham, UK, UK, 2000.