

Build Android Apps
with Python



Pro
Android Python
with SL4A

Paul Ferrill

Apress®

Pro Android Python with SL4A



Paul Ferrill

Apress®

Pro Android Python with SL4A

Copyright © 2011 by Paul Ferrill

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN 978-1-4302-3569-9

ISBN 978-1-4302-3570-5 (eBook)

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

President and Publisher: Paul Manning

Lead Editor: Tom Welsh

Technical Reviewer: Justin Grammens

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editors: Mary Tobin, Corbin Collins

Copy Editor: Nancy Sixsmith

Production Support: Patrick Cunningham

Indexer: SPI Global

Artist: April Milne

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at www.apress.com/bulk-sales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

*To my wife, Sandy, for your tireless support of me and our family. I could not have done this without you.
And to my wonderful children who put up with a preoccupied daddy for way too long.*

—Paul Ferrill

Contents at a Glance

About the Author	xi
About the Technical Reviewer	xii
Acknowledgments	xiii
Preface	xiv
■ Chapter 1: Introduction	1
■ Chapter 2: Getting Started	27
■ Chapter 3: Navigating the Android SDK.....	57
■ Chapter 4: Developing with Eclipse	83
■ Chapter 5: Exploring the Android API	113
■ Chapter 6: Background Scripting with Python	139
■ Chapter 7: Python Scripting Utilities	165
■ Chapter 8: Python Dialog Box–based GUIs	195
■ Chapter 9: Python GUIs with HTML.....	221
■ Chapter 10: Packaging and Distributing.....	249
Index.....	273

Contents

About the Author	xi
About the Technical Reviewer	xii
Acknowledgments	xiii
Preface	xiv
Chapter 1: Introduction	1
Why SL4A?.....	1
The World of Android	2
Android Application Anatomy	2
Activities	3
Intents.....	3
SL4A History	4
SL4A Architecture	4
SL4A Concepts.....	6
JavaScript Object Notation (JSON)	6
Events	7
Languages	7
Beanshell 2.0b4.....	7
Lua 5.1.4.....	8
Perl 5.10.1	9
PHP 5.3.3.....	11
Rhino 1.7R2	11

JRuby 1.4.....	12
Shell.....	13
Python.....	13
Summary	26
Chapter 2: Getting Started	27
Installing SL4A on the Device	27
Installing the Android SDK.....	39
Linux.....	39
Mac OS X.....	41
Windows.....	41
Installing Python	43
Remotely Connecting to the Device.....	45
Device Settings.....	49
Executing Simple Programs.....	51
Summary	55
Chapter 3: Navigating the Android SDK.....	57
Wading Through the SDK Documentation.....	57
Examining the Different SDK Components	59
Testing With the Android Emulator	60
Android Debug Bridge.....	68
Dalvik Debug Monitor Service (DDMS)	79
Summary	81
Chapter 4: Developing with Eclipse	83
Installing Eclipse on a Development Machine	83
Eclipse Basics.....	88
Perspectives	90
Projects.....	93

Android Development Toolkit.....	94
Using Pydev	99
Using Multiple File Types in Eclipse	107
Summary	110
Chapter 5: Exploring the Android API	113
Exploring the Android APIs	115
Android Facades	116
ActivityResultFacade	118
AndroidFacade.....	118
ApplicationManagerFacade	121
BatteryManagerFacade	121
BluetoothFacade.....	123
CameraFacade.....	123
CommonIntentsFacade.....	124
ContactsFacade	125
EventFacade	127
EyesFreeFacade	127
LocationFacade	127
MediaPlayerFacade	128
MediaRecorderFacade.....	128
PhoneFacade	128
PreferencesFacade.....	128
SensorManagerFacade.....	129
SettingsFacade.....	130
SignalStrengthFacade	130
SmsFacade.....	131
SpeechRecognitionFacade	132
TextToSpeechFacade	132
ToneGeneratorFacade	132

UiFacade.....	132
WakeLockFacade.....	137
WebCamFacade.....	137
WifiFacade.....	137
Summary	138
Chapter 6: Background Scripting with Python	139
Background Tasks	139
Triggers	141
Orientation-based Actions	142
Location-based Actions	145
Time-based Actions	146
Elapsed Time-based Triggers	148
FTP File Sync Tool.....	150
Syncing Photos with Flickr	158
Syncing with Google Docs	160
A Startup Launcher	162
Summary	164
Chapter 7: Python Scripting Utilities	165
Python Libraries.....	165
E-mail-Based Applications.....	168
Location-Aware Applications.....	172
Tweet My Location	172
Killing a Running App	186
URL File Retriever	188
Python FTP Server	190
Summary	194

Chapter 8: Python Dialog Box–based GUIs	195
UI Basics	195
Book Title Search.....	199
Convenience Dialog Boxes	201
Progress Dialog Boxes.....	203
Modal versus Non–Modal Dialog Boxes	205
Options Menu	207
File Listing with dialogCreateAlert.....	207
Dialog Boxes as Python Objects	209
Podplayer App	213
Building the mysettings App.....	216
Summary	220
Chapter 9: Python GUIs with HTML.....	221
HTML and Basic Information Display.....	221
HTML and JavaScript.....	224
HTML GUI Form Basics	226
Simple HTML Forms	228
Cascading Style Sheets	230
SMS Merger.....	233
Summary	247
Chapter 10: Packaging and Distributing.....	249
QR Codes	249
Application Packages	251
Packaging Your Own Application.....	264
Building with Ant	264
Compiling SL4A.....	266
Finishing Touches.....	269

Winding Down.....	271
Summary	271
Index.....	273

About the Author



■ **Paul Ferrill** has a BS and MS in electrical engineering and has been writing about computers for more than 25 years. He currently serves as CTO for Avionics Test and Analysis Corporation, working on multiple DoD projects. Software development has been his primary focus, along with architecting large-scale data management and storage systems. He also serves on several DoD standards committees, providing input to the next generation of data recording and transmission standards.

He has a long history with both Microsoft and open source technologies. His two favorite languages are Visual Basic and Python. He's had articles published in *PC Magazine*, *PC Computing*, *InfoWorld*, *Computer World*, *Network World*, *Network Computing*, *Federal Computer Week*, *Information Week*, and multiple web sites.

About the Technical Reviewer



Justin Grammens has been writing software for 12 years, holds a masters degree in Software Systems, and has a patent pending on the process of a system to collect and rate digital media. He has written applications for a variety of mobile platforms in a number of different market sectors and is the cofounder of Recursive Awesome, LLC; owner of Localtone, LLC; and founder of Mobile Twin Cities.

Justin has built online e-commerce systems, real-time mapping solutions, large-scale tax accounting software, and technology for Internet radio stations. Having worked with Android since version 1.0, Justin has spoken on mobile technology at conferences and software development groups since 2008.

Justin has developed Android applications for Best Buy, McDonald's, BuzzFeed, and Consolidated Knowledge; and is co-creator of a cross-platform streaming video service called Mobile Vidhub. Justin is employed by Code 42 as a Director of Mobile Technology and lives in St. Paul, MN, with his wife.

Acknowledgments

I would like to acknowledge the excellent staff at Apress who managed to get this book completed on time through multiple delays and reworking of the original title. You've made the process much less frightening for a first-time author than I expected.

A special thanks goes to coordinating editors Mary Tobin and Corbin Collins, and to Tom Welsh, the lead editor.

I'd also like to thank Frank Pohlmann for convincing me to do this project in the first place.

Thank you to the technical reviewer, Justin Grammens, for a keen set of eyes and helpful comments. A big thank you to Robbie Matthews, who has become one of the primary contributors to the SL4A project and provided help when things didn't make sense.

Thanks also to the folks at TechSmith, and Betsy Weber in particular, for their fantastic Snagit product, without which the screenshots would have been so much harder.

Final thanks go to my son Micah Ferrill for his help with the Python code.

Preface

It's no secret that traditional computing patterns are undergoing a radical change. The proliferation of smartphones with ever-increasing processing power will only accelerate the process. Tablet devices have seen a much broader adoption as extensions of the smartphone platform where previous attempts to downsize general-purpose computers failed. While the operating system of the most popular mobile devices may be different from the user's perspective, it has more in common with a desktop system than you might think.

Google's Android platform has seen a huge increase over the last year and is challenging Apple's iOS for market share. Apple's wide lead in applications has been steadily dwindling although the jury is still out when it comes to quality. Building those applications has, for the most part, been restricted to Objective C for iOS and Java for Android. There are a few other options if you take into consideration the MonoTouch and MonoDroid projects, but that's about it.

Mobile devices will probably never completely replace traditional computers, although the division of activity will continue to swing toward the one you have access to the most. This book is about bringing some of the flexibility you get with a desktop computer in the form of writing simple programs or scripts to accomplish a specific task. I know I've learned a lot along the way, and it is my sincere hope that through reading this book you will glean a thing or two as well.

CHAPTER 1

Introduction

This book is about writing real-world applications for the Android platform primarily using the Python language and a little bit of JavaScript. While there is nothing wrong with Java, it really is overkill when all you need to do is turn on or off a handful of settings on your Android device. The Scripting Layer for Android (SL4A) project was started to meet that specific need. This book will introduce you to SL4A and give you the power to automate your Android device in ways you never thought possible.

Why SL4A?

One of the first questions you probably have about this book is, “Why would I want to use SL4A instead of Java?” There are several answers to that question. One is that not everyone is a fan of Java. The Java language is too heavyweight for some and is not entirely open source. It also requires the use of an edit / compile / run design loop that can be tedious for simple applications. An equally legitimate answer is simply “I want to use X”, where X could be any number of popular languages.

Google provides a comprehensive software development kit (SDK) aimed specifically at Java developers, and most applications available from the Android market are probably written in Java. I’ll address the Android SDK in Chapter 3 and use a number of the tools that come with it throughout the book.

■ **Note** SL4A currently supports Beanshell, JRuby, Lua, Perl, PHP, Python, and Rhino.

SL4A is really targeted at anyone looking for a way to write simple scripts to automate tasks on an Android device using any of the supported languages, including Java through Beanshell. It provides an interactive console in which you can type in a line of code and immediately see the result. It even makes it possible, in many cases, to reuse code you’ve written for a desktop environment. The bottom line is that SL4A makes it possible both to write code for Android-based devices in languages other than Java and to do it in a more interactive way.

The World of Android

Google jumped into the world of mobile operating systems in a big way when it bought Android, Inc. in 2005. It's really pretty amazing how far it has come in such a short time. The Android community is huge and has spawned a wide range of conferences, books, and support materials that are easily available over the Internet.

This is a good point to define a few terms that you'll see throughout the rest of this book. Android applications are typically packaged into `.apk` files. These are really just `.zip` files containing everything needed by the application. In fact, if you rename an `.apk` file to `.zip`, you can open it with any archive tool and examine the contents.

Most Android devices come from the manufacturer with the systems files protected to prevent any inadvertent or malicious manipulation. The Android operating system (OS) is essentially Linux at the core and provides much of the same functionality you would find on any Linux desktop. There are ways to unlock the system areas and provide *root*, or unrestricted, access to the entire filesystem on an Android device. This process is appropriately called *rooting* your device, and once complete, the device is described as *rooted*. SL4A does not require a rooted device, but will work on one if you have chosen this path.

Android Application Anatomy

Android is based on the Linux operating system (at the time of writing, version 2.6 of the Linux kernel). Linux provides all the core plumbing such as device drivers, memory and process management, network stack, and security. The kernel also adds a layer of abstraction between the hardware and applications. To use an anatomical analogy, you might think of Linux as the skeleton, muscles, and organs of the Android body.

The next layer up the Android stack is the Dalvik Virtual Machine (DVM). This piece provides the core Java language support and most of the functionality of the Java programming language. The DVM is the brains in which the majority of all processing takes place. Every Android application runs in its own process space in a private instance of the DVM. The application framework provides all the necessary components needed by an Android application. From the Google Android documentation:

“Developers have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components. Any application can publish its capabilities, and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This same mechanism allows components to be replaced by the user.

Underlying all applications is a set of services and systems, including:

- *A rich and extensible set of Views that can be used to build an application, including lists, grids, text boxes, buttons, and even an embeddable web browser*
- *Content Providers that enable applications to access data from other applications (such as Contacts) or to share their own data*
- *A Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files*

- *A Notification Manager that enables all applications to display custom alerts in the status bar*
- *An Activity Manager that manages the lifecycle of applications and provides a common navigation backstack”¹*

All Android applications are based on three core components: activities, services, and receivers. These core components are activated through messages called *intents*. SLAA gives you access to much of the core Android functionality through its API facade, so it’s a good idea to understand some of the basics. Chapters 3 and 5 look at the Android SDK and Android application programming interface (API) in detail, so I’ll save the specifics for later. For now, I’ll introduce you to activities and intents, as they will be used extensively.

Activities

The Android documentation defines an *activity* as “an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an e-mail, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen but may be smaller than the screen and float on top of other windows.”

Android applications consist of one or more activities loosely coupled together. Each application will typically have a “main” activity that can, in turn, launch other activities to accomplish different functions.

Intents

From the Google documentation: “An intent is a simple message object that represents an *intention* to do something. For example, if your application wants to display a web page, it expresses its *intent* to view the URI by creating an intent instance and handing it off to the system. The system locates some other piece of code (in this case, the browser) that knows how to handle that intent and runs it. Intents can also be used to broadcast interesting events (such as a notification) system-wide.”

An intent can be used with `startActivity` to launch an *activity*, `broadcastIntent` to send it to any interested `BroadcastReceiver` components, and `startService(Intent)` or `bindService(Intent, ServiceConnection, int)` to communicate with a background service. Intents use primary and secondary attributes that you must provide in the form of arguments.

There are two primary attributes:

- **action:** The general action to be performed, such as `VIEW_ACTION`, `EDIT_ACTION`, `MAIN_ACTION`, and so on
- **data:** The data to operate on, such as a person record in the contacts database, expressed as a Uniform Resource Identifier (URI)

¹ <http://developer.android.com/guide/basics/what-is-android.html>

There are four types of secondary attributes:

- **category:** Gives additional information about the action to execute. For example, `LAUNCHER_CATEGORY` means it should appear in the Launcher as a top-level application, while `ALTERNATIVE_CATEGORY` means it should be included in a list of alternative actions the user can perform on a piece of data.
- **type:** Specifies an explicit type (a MIME type) of the intent data. Normally, the type is inferred from the data itself. By setting this attribute, you disable that evaluation and force an explicit type.
- **component:** Specifies an explicit name of a component class to use for the intent. Normally this is determined by looking at the other information in the intent (the action, data/type, and categories) and matching that with a component that can handle it. If this attribute is set, none of the evaluation is performed, and this component is used exactly as is. By specifying this attribute, all the other intent attributes become optional.
- **extras:** A bundle of any additional information. This can be used to provide extended information to the component. For example, if we have an action to send an e-mail message, we could also include extra pieces of data here to supply a subject, body, and so on.

SL4A History

SL4A was first announced on the Google Open Source blog in June of 2009 and was originally named Android Scripting Environment (ASE). It was primarily through the efforts of Damon Kohler that this project came to see the light of day. Others have contributed along the way as the project has continued to mature. The most recent release as of this writing is r4, although you'll also find experimental versions available on the SL4A web site (<http://code.google.com/p/android-scripting>).

SL4A Architecture

At its lowest level, SL4A is essentially a scripting host, which means that as an application it hosts different interpreters each of which processes a specific language. If you were to browse the SL4A source code repository, you would see a copy of the source tree of each language. This gets cross-compiled for the ARM architecture using the Android Native Development Kit (NDK) and loads as a library when SL4A launches a specific interpreter. At that point, the script will be interpreted line by line.

The basic architecture of SL4A is similar to what you would see in a distributed computing environment. Figure 1-1 shows in pictorial form the flow of execution when you launch SL4A and then run a script (in this case, `hello.py`). Every SL4A script must import or source an external file, such as `android.py` for Python, which will define a number of proxy functions needed to communicate with the Android API.

The actual communication between SL4A and the underlying Android operating system uses a remote procedure call (RPC) mechanism and JavaScript Object Notation (JSON). You normally find RPC used in a distributed architecture in which information is passed between a client and a server. In the case of SL4A, the server is the Android OS, and the client is an SL4A script. This adds a layer of separation between SL4A and the Android OS to prevent any malicious script from doing anything harmful.

Security is a concern and is one of the reasons that SL4A uses the RPC mechanism. Here's how the SL4A wiki describes it:

“RPC Authentication: SL4A enforces per-script security sandboxing by requiring all scripts to be authenticated by the corresponding RPC server. In order for the authentication to succeed, a script has to send the correct handshake secret to the corresponding server. This is accomplished by:

1. reading the `AP_HANDSHAKE` environment variable.
2. calling the RPC method `_authenticate` with the value of `AP_HANDSHAKE` as an argument.

The `_authenticate` method must be the first RPC call and should take place during the initialization of the Android library. For example, see Rhino's or Python's Android module”².

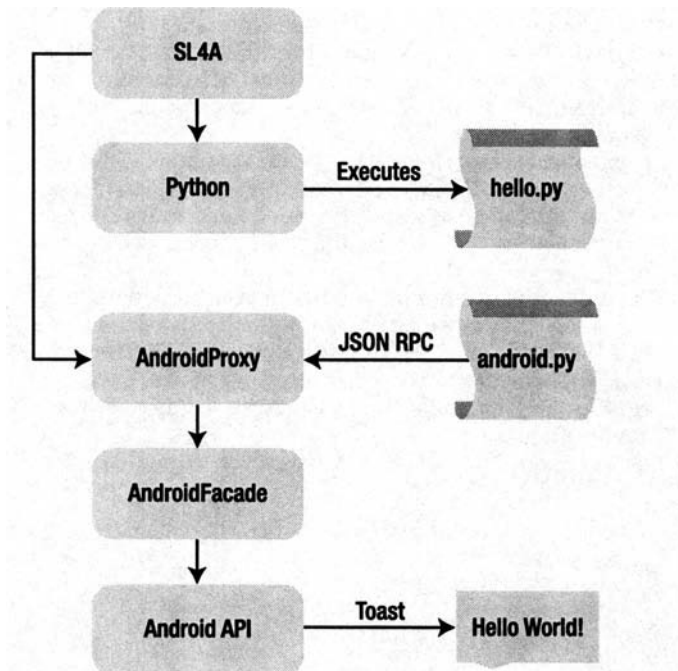


Figure 1-1. SL4A execution flow diagram

² <http://code.google.com/p/android-scripting/wiki/InterpreterDeveloperGuide>

SL4A Concepts

There are a number of concepts used by SL4A that need to be introduced before we actually use them. At a very high level, SL4A provides a number of functional pieces working in concert together. Each supported language has an interpreter that has been compiled to run on the Android platform. Along with the interpreters is an abstraction layer for the Android API. This abstraction layer provides a calling interface in a form expected for each language. The actual communication between the interpreters and the native Android API uses inter-process communication (IPC) as an extra layer of protection. Finally, there is support for an on-device environment to test scripts interactively.

Although Figure 1-1 shows Python as the interpreter, the concept works pretty much the same for all supported languages. Each interpreter executes the language in its own process until an API call is made. This is then passed along to the Android OS using the RPC mechanism. All communication between the interpreter and the Android API typically uses JSON to pass information.

JavaScript Object Notation (JSON)

SL4A makes heavy use of JSON to pass information around. You might want to visit the <http://www.json.org> web site if you've never seen JSON before. In its simplest form JSON is just a way of defining a data structure or an object in much the same way you would in the context of a program. For the most part, you will see JSON structures appear as a series of name/value pairs. The name part will always be a string while the value can be any JavaScript object.

In SL4A, you will find that many of the API calls return information using JSON. Fortunately, there are multiple options when it comes to creating, parsing, and using JSON. Python treats JSON as a first-class citizen with a full library of tools to convert from JSON to other native Python types and back again. The Python Standard Library `pprint` module is a convenient way to display the contents of a JSON response in a more readable format.

The Python Standard Library includes a JSON module with a number of methods to make handling JSON much easier. Because JSON objects can contain virtually any type of data, you must use encoders and decoders to get native Python data types into a JSON object. This is done with the `json.JSONEncoder` and `json.JSONDecoder` methods. When you move a JSON object from one place to another, you must serialize and then deserialize that object. This requires the `json.load()` and `json.loads()` functions for decoding, and `json.dump()` plus `json.dumps()` for encoding.

There are a large number of web services that have adopted JSON as a standard way to implement an API. Here's one from Yahoo for images:

```
{
  "Image": {
    "Width":800,
    "Height":600,
    "Title":"View from 15th Floor",
    "Thumbnail":
      {
        "Url":"http://s.cdn.mm-b1.yimg.com/image/481989943",
        "Height": 125,
        "Width": "100"
      },
    "IDs":[ 116, 943, 234, 38793 ]
  }
}
```

Events

The Android OS uses an event queue as a means of handling specific hardware-generated actions such as when the user presses one of the hardware keys. Other possibilities include any of the device sensors such as the accelerometer, GPS receiver, light sensor, magnetometer, and touch screen. Each sensor must be explicitly turned on before information can be retrieved.

The SL4A API facade provides a number of API calls that will initiate some type of action resulting in an event. These include the following:

- `startLocating()`
- `startSensing()`
- `startTrackingPhoneState()`
- `startTrackingSignalStrengths()`

Each of these calls will begin gathering some type of data and generate an event such as a “location” event or a “phone” event. Any of the supported languages can register an event handler to process each event. The `startLocating()` call takes two parameters, allowing you to specify the minimum distance and the minimum time between updates.

Languages

One of the things that SL4A brings to the table is lots of language choices. As of the writing of this book, those choices include Beanshell, Lua, JRuby, Perl, PHP, Python, and Rhino (versions given in the following sections). You can also write or reuse shell scripts if you like. Without question, the most popular of all these languages is Python. Support for the others has not been near the level of Python, up to this point, but it is possible to use them if you’re so inclined.

Beanshell 2.0b4

Beanshell is an interesting language in that it’s basically interpreted Java. It kind of begs the question of why you would want an interpreted Java when you could just write native Java using the Android SDK. The Beanshell interpreter does provide an interactive tool to write and test code. It’s definitely not going to be the fastest code, but you might find it useful for testing code snippets without the need to go through the whole compile/deploy/test cycle.

Examining the `android.bsh` file shows the code used to set up the JSON data structures for passing information to and receiving information from the Android OS. Here’s what the basic call function looks like:

```
call(String method, JSONArray params) {
    JSONObject request = new JSONObject();
    request.put("id", id);
    request.put("method", method);
    request.put("params", params);
    out.write(request.toString() + "\n");
    out.flush();
    String data = in.readLine();
}
```

```

    if (data == null) {
        return null;
    }
    return new JSONObject(data);
}

```

Here's a simple `hello_world.bsh` script:

```

source("/sdcard/com.googlecode.bshforandroid/extras/bsh/android.bsh");
droid = Android();
droid.call("makeToast", "Hello, Android!");

```

Lua 5.1.4

Lua.org describes Lua as “an extension programming language designed to support general procedural programming with data description facilities”.³ The term *extension programming language* means that Lua is intended to be used to extend an existing program through scripting. This fits in well with the concept of SL4A.

From a syntax perspective, Lua resembles Python somewhat in that it doesn't use curly braces to wrap code blocks or require a semicolon for statement termination, although you can do this if you want to. In the case of a function definition, Lua uses the reserved word `function` to begin the code block and then the reserved word `end` to mark the end.

Lua has most of the standard data types you would expect in a modern language and also includes the concept of a table. In Lua, a table is a dynamically created object that can be manipulated much like pointers in conventional languages. Tables must be explicitly created before use. Tables can also refer to other tables, making them well suited to recursive data types. The list of generic functions for manipulating tables includes `table.concat`, `.insert`, `.maxn`, `.remove`, and `.sort`.

From the Lua web site, here's a short Lua code snippet that creates a circular linked list:

```

list = {}                -- creates an empty table
current = list
i = 0
while i < 10 do
    current.value = i
    current.next = {}
    current = current.next
    i = i+1
end
current.value = i
current.next = list

```

³ <http://www.lua.org/manual/5.1/manual.html>

Here's the Lua code that implements the RPC call function:

```
function rpc(client, method, ...)
  assert(method, 'method param is nil')
  local rpc = {
    ['id'] = id,
    ['method'] = method,
    params = arg
  }
  local request = json.encode(rpc)
  client:send(request .. '\n')
  id = id + 1
  local response = client:receive('*l')
  local result = json.decode(response)
  if result.error ~= nil then
    print(result.error)
  end
  return result
end
```

The obligatory Lua hello world script:

```
require "android"

name = android.getInput("Hello!", "What is your name?")
android.printDict(name) -- A convenience method for inspecting dicts (tables).
android.makeToast("Hello, " .. name.result)
```

The Lua wiki has links to sample code with a large number of useful snippets.

Perl 5.10.1

Perl probably qualifies as the oldest of the languages available in SL4A if you don't count the shell. It dates back to 1987 and has been used in just about every type of computing application you can think of. The biggest advantage of using Perl is the large number of code examples to draw from. Coding the `hello_world.pl` script looks a lot like that of other languages:

```
use Android;
my $a = Android->new();
$a->makeToast("Hello, Android!");
```

Here's the Perl code needed to launch an SL4A script:

```
# Given a method and parameters, call the server with JSON,
# and return the parsed the response JSON. If the server side
# looks to be dead, close the connection and return undef.
sub do_rpc {
  my $self = shift;
  if ($self->trace) {
    show_trace(qq[do_rpc: $self: @_]);
  }
}
```



```

my $method = pop;
my $request = to_json({ id => $self->{id},
                      method => $method,
                      params => [ @_ ] });
if (defined $self->{conn}) {
    print { $self->{conn} } $request, "\n";
    if ($self->trace) {
        show_trace(qq[client: sent: "$request"]);
    }
    $self->{id}++;
    my $response = readline($self->{conn});
    chomp $response;
    if ($self->trace) {
        show_trace(qq[client: rcvd: "$response"]);
    }
    if (defined $response && length $response) {
        my $result = from_json($response);
        my $success = 0;
        my $error;
        if (defined $result) {
            if (ref $result eq 'HASH') {
                if (defined $result->{error}) {
                    $error = to_json( { error => $result->{error} } );
                } else {
                    $success = 1;
                }
            } else {
                $error = "illegal JSON reply: $result";
            }
        }
        unless ($success || defined $error) {
            $error = "unknown JSON error";
        }
        if (defined $error) {
            printf STDERR "$0: client: error: %s\n", $error;
        }
        if ($Opt{trace}) {
            print STDERR Data::Dumper->Dump([$result], [qw(result)]);
        }
        return $result;
    }
}
$self->close;
return;
}

```

PHP 5.3.3

PHP is, without a doubt, one of the most successful general-purpose scripting languages for creating dynamic web pages. From humble beginnings as the Personal Home Page, the acronym PHP now stands for PHP: Hypertext Preprocessor. PHP is a free and open source language with implementations for virtually every major operating system available free of charge.

Here's the PHP code needed to launch an SL4A script via an RPC:

```
public function rpc($method, $args)
{
    $data = array(
        'id'=>$this->_id,
        'method'=>$method,
        'params'=>$args
    );
    $request = json_encode($data);
    $request .= "\n";
    $sent = socket_write($this->_socket, $request, strlen($request));
    $response = socket_read($this->_socket, 1024, PHP_NORMAL_READ) or die("Could not
read input\n");
    $this->_id++;
    $result = json_decode($response);

    $ret = array ('id' => $result->id,
        'result' => $result->result,
        'error' => $result->error
    );
    return $ret;
}
```

The PHP version of `hello_world.php` looks like this:

```
<?php
require_once("Android.php");
$droid = new Android();
$name = $droid->getInput("Hi!", "What is your name?");
$droid->makeToast('Hello, ' . $name['result']);
```

You get a number of other example scripts when you install PHP along with the basic `hello_world.php`.

Rhino 1.7R2

The Rhino interpreter gives you a way to write stand-alone JavaScript code. JavaScript is actually standardized as ECMAScript under ECMA-262. You can download the standard from <http://www.ecma-international.org/publications/standards/Ecma-262.htm>. The advantages of having a JavaScript interpreter are many. If you plan on building any type of custom user interface using HTML and JavaScript, you could prototype the JavaScript part and test it with the Rhino interpreter.

The `android.js` file for Rhino resembles that of the other languages in many aspects. Here's what the RPC call definition looks like:

```
this.rpc = function(method, args) {
  this.id += 1;
  var request = JSON.stringify({'id': this.id, 'method': method,
                              'params': args});
  this.output.write(request + '\n');
  this.output.flush();
  var response = this.input.readLine();
  return eval("(" + response + ")");
},
```

Here's a simple Rhino `hello_world.js` script:

```
load("/sdcard/s14a/extras/rhino/android.js");
var droid = new Android();
droid.makeToast("Hello, Android!");
```

JRuby 1.4

One of the potential hazards of any open source project is neglect. At the time of this writing, based on SL4A r4, the JRuby interpreter has suffered from neglect and doesn't even run the `hello_world.rb` script. In any case, here's what that script looks like:

```
require "android"
droid = Android.new
droid.makeToast "Hello, Android!"
```

The JRuby interpreter does launch, and you can try out some basic JRuby code with it. Here's what the Android class looks like in Ruby:

```
class Android

  def initialize()
    @client = TCPSocket.new('localhost', AP_PORT)
    @id = 0
  end

  def rpc(method, *args)
    @id += 1
    request = {'id' => @id, 'method' => method, 'params' => args}.to_json()
    @client.puts request
    response = @client.gets()
    return JSON.parse(response)
  end

  def method_missing(method, *args)
    rpc(method, *args)
  end

end
```

Shell

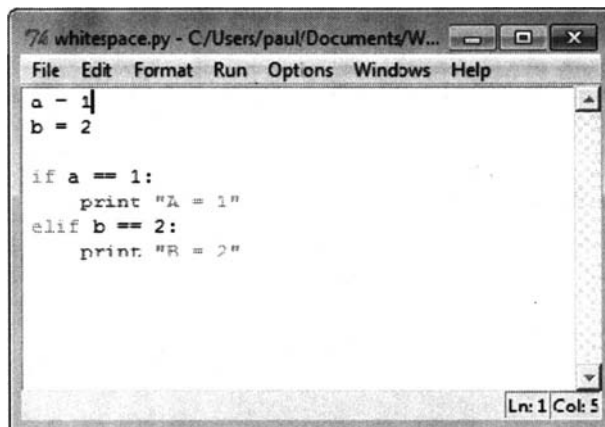
If you're a shell script wizard, then you'll feel right at home with SL4A's shell interpreter. It's essentially the same bash script environment you would see at a typical Linux terminal prompt. You'll find all the familiar commands for manipulating files like `cp`, `ls`, `mkdir`, and `mv`.

Python

Python has a wide usage and heavy following, especially within Google. In fact, its following is so significant they hired the inventor of the language, Guido van Rossum. Python has been around for quite a while and has many open source projects written in the language. It also has seen the most interest as far as SL4A is concerned, so you'll find more examples and discussions in the forums than for any of the other languages. For that reason, I will spend a little more time introducing the language, trying to hit the highlights of things that will be important from an SL4A perspective.

Language Basics

Knowing the Python language is not an absolute requirement for this book, but it will help. The first thing you need to know about Python is that everything is an object. The second thing is that whitespace is meaningful in Python. By that, I mean Python uses either tabs or actual spaces (ASCII 32) instead of curly braces to control code execution (see Figure 1-2). Third, it's important to remember that Python is a case-sensitive language.



```

74 whitespace.py - C:/Users/paul/Documents/W...
File Edit Format Run Options Windows Help
a = 1
b = 2

if a == 1:
    print "A = 1"
elif b == 2:
    print "B = 2"

Ln: 1 Col: 5
  
```

Figure 1-2. Example of whitespace usage in Python

Python is a great language to use when teaching an “introduction to computer programming” course. Every installation of standard Python comes with a command-line interpreter where you can type in a line of code and immediately see the result. To launch the interpreter, simply enter `python` at a command prompt (Windows) or terminal window (Linux and Mac OS X). At this point, you should see a few lines with version information followed by the triple arrow prompt (`>>>`), letting you know that you're inside the Python interpreter as shown here:

```
C:\Users\paul>python
Python 2.6.6 (r266:84297, Aug 24 2010, 18:13:38) [MSC v.1500 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python uses a number of naming conventions that you will see if you examine much Python code. The first is the double underscore, which is used in Python to “mangle” or change names as a way to define private variables and methods used inside a class. You will see this notation used for “special” methods such as `__init__(self)`. If a class has the special `__init__` method, it will be invoked whenever a new instantiation of that class occurs.

For example:

```
>>> class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> xy = Point(1,2)
>>> xy.x, xy.y
(1, 2)
```

As you can see from the example, `self` is used as a reserved word in Python and refers to the first argument of a method. It’s actually a Python convention and, in reality, has no special meaning to Python. However, because it’s a widely accepted convention, you’ll want to stick with it to avoid any potential issues. Technically, `self` is a reference to the class or function itself. Methods within a class may call other methods in the same class by using the method attributes of the `self` argument.

Python has a short list of built-in constants. The main ones you’ll run into are `False`, `True`, and `None`. `False` and `True` are of type `bool` and show up primarily in logical tests or to create an infinite loop.

One of the things that frequently confuses new users of the language is the variety of data types. The following sections give quick overview of the key data types you’ll need to use Python and SL4A.

Dictionary: An Unordered Set of Key/Value Pairs Requiring Unique Keys

A Python dictionary maps directly to a JSON data structure. The syntax for defining a dictionary uses curly braces to enclose entries and a colon between the key and value. Here’s what a simple dictionary definition looks like:

```
students = {'barney' : 1001, 'betty' : 1002, 'fred' : 1003, 'wilma' : 1004}
```

To reference entries, use the key, as shown here:

```
students['barney'] = 999
students['betty'] = 1000
```

You can also use the `dict()` constructor to build dictionaries. When the key is a simple string, you can create a new dictionary using arguments passed to `dict()` such as the following:

```
students = dict(barney=1001, betty=1002, fred=1003, wilma=1004)
```

Because everything in Python is an object, you can expect to see methods associated with a dictionary object. As you might expect, there are methods to return the keys and the values from a dictionary. Here's what that would look like for the `students` dictionary:

```
>>> students
{'barney': 1001, 'betty': 1002, 'fred': 1003, 'wilma': 1004}
>>> students.keys()
['barney', 'betty', 'fred', 'wilma']
>>> students.values()
[1001, 1002, 1003, 1004]
```

The square bracket convention denotes a list. Evaluating the `students.keys()` statement returns a list of keys from the `students` dictionary.

List: A Built-In Python Sequence Similar to an Array in Other Languages

In Python, a *sequence* is defined as “an iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence.” An *iterable* is defined as “a container object capable of returning its members one at a time.” Python provides direct language support for iteration because it's one of the more common operations in programming. List objects provide a number of methods to make working with them easier. From the Python documentation:

- `list.append(x)`: Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.
- `list.extend(L)`: Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.
- `list.insert(I,x)`: Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`: Remove the first item from the list whose value is `x`. It is an error if there is no such item.
- `list.pop([i])`: Remove the item at the given position in the list and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position.) You will see this notation frequently in the Python Library Reference.
- `list.index(x)`: Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.
- `list.count(x)`: Return the number of times `x` appears in the list.
- `list.sort`: Sort the items of the list, in place.
- `list.reverse(x)`: Reverse the elements of the list, in place.