python-for-android Documentation

Release 0.1

Alexander Taylor

October 06, 2015

Contents

1 Contents **3**
- 1.1 Quickstart . . . 3
- 1.2 Commands . . . 7
- 1.3 Recipes . . . 7
- 1.4 Bootstraps . . . 14
- 1.5 Accessing Android APIs . . . 16
- 1.6 Troubleshooting . . . 17
- 1.7 Differences to the old python-for-android project . . . 18
- 1.8 Contributing . . . 19
- 1.9 Old p4a toolchain doc . . . 19
- 1.10 Indices and tables . . . 43

2 Indices and tables **45**

Python Module Index **47**

python-for-android is an open source build tool to let you package Python code into standalone android APKs that can be passed around, installed, or uploaded to marketplaces such as the Play Store just like any other Androip app. This tool was originally developed for the Kivy cross-platform graphical framework, but now supports multiple bootstraps and can be easily extended to package other types of Python app for Android.

python-for-android supports two major operations; first, it can compile the Python interpreter, its dependencies, back-end libraries and python code for Android devices. This stage is fully customisable, you can install as many or few components as you like. The result is a standalone Android project which can be used to generate any number of different APKs, even with different names, icons, Python code etc. The second function of python-for-android is to provide a simple interface to these distributions, to generate from such a project a Python APK with build parameters and Python code to taste.

CHAPTER 1

Contents

1.1 Quickstart

These simple steps run through the most simple procedure to create an APK with some simple default parameters. See the commands documentation for all the different commands and build options available.

Warning: These instructions are quite preliminary. The installation and use process will become more standard in the near future.

1.1.1 Installation

The easiest way to install is with pip. You need to have setuptools installed, then run:

```
pip install git+https://github.com/kivy/python-for-android.git
```

This should install python-for-android (though you may need to run as root or add –user).

You could also install python-for-android manually, either via git:

```
git clone https://github.com/kivy/python-for-android.git
cd python-for-android
```

Or by direct download:

```
wget https://github.com/kivy/python-for-android/archive/master.zip
unzip revamp.zip
cd python-for-android-revamp
```

Then in both cases run `python setup.py install`.

1.1.2 Dependencies

python-for-android has several dependencies that must be installed, via your package manager or otherwise. These include:

- git
- ant
- python2
- the Android SDK and NDK (see below)

- a Java JDK (e.g. openjdk-7)
- zlib (including 32 bit)
- libncurses (including 32 bit)
- unzip
- ccache (optional)

On recent versions of Ubuntu and its derivatives you may be able to install all many of these with:

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install -y build-essential ccache git zlib1g-dev python2.7 python2.7-dev libncurses5:i38
```

When installing the Android SDK and NDK, note the filepaths where they may be found, and the version of the NDK installed. You may need to set environment variables pointing to these later.

1.1.3 Basic use

python-for-android provides two executables, `python-for-android` and `p4a`. These are identical and interchangeable, you can substitute either one for the other. These instructions all use `python-for-android`.

You can test that p4a was installed correctly by running `python-for-android recipes`. This should print a list of all the recipes available to be built into your APKs.

Before running any apk packaging or distribution creation, it is essential to set some env vars. Make sure you have installed the Android SDK and NDK, then:

- Set the `ANDROIDSDK` env var to the `/path/to/the/sdk`
- Set the `ANDROIDNDK` env var to the `/path/to/the/ndk`
- Set the `ANDROIDAPI` to the targeted API version (or leave it unset to use the default of `14`).
- Set the `ANDROIDNDKVER` env var to the version of the NDK downloaded, e.g. the current NDK is `r10e` (or leave it unset to use the default of `r9`.

This is **NOT** the only way to set these variables, see the *setting SDK/NDK paths* section for other options and their details.

To create a basic distribution, run .e.g:

```
python-for-android create --dist_name=testproject --bootstrap=pygame \
    --requirements=sdl,python2
```

This will compile the distribution, which will take a few minutes, but will keep you informed about its progress. The arguments relate to the properties of the created distribution; the dist_name is an (optional) unique identifier, and the requirements is a list of any pure Python pypi modules, or dependencies with recipes available, that your app depends on. The full list of builtin internal recipes can be seen with `python-for-android recipes`.

Note: Compiled dists are not located in the same place as with old python-for-android, but instead in an OS-dependent location. The build process will print this location when it finishes, but you no longer need to navigate there manually (see below).

To build an APK, use the `apk` command:

```
python-for-android apk --private /path/to/your/app --package=org.example.packagename \
    --name="Your app name" --version=0.1
```

The arguments to `apk` can be anything accepted by the old python-for-android build.py; the above is a minimal set to create a basic app. You can see the list with `python-for-android apk help`.

A new feature of python-for-android is that you can do all of this with just one command:

```
python-for-android apk --private /path/to/your/app \
    --package=org.example.packagename --name="Your app name" --version=0.5
    --bootstrap=pygame --requirements=sdl,python2 --dist_name=testproject
```

This combines the previous `apk` command with the arguments to `create`, and works in exactly the same way; if no internal distribution exists with these requirements then one is first built, before being used to package the APK. When the command is run again, the build step is skipped and the previous dist re-used.

Using this method you don't have to worry about whether a dist exists, though it is recommended to use a different `dist_name` for each project unless they have precisely the same requirements.

You can build an SDL2 APK similarly, creating a dist as follows:

```
python-for-android create --dist_name=testsdl2 --bootstrap=sdl2 --requirements=sdl2,python2
```

You can then make an APK in the same way, but this is more experimental and doesn't support as much customisation yet.

There is also experimental support for building APKs with Vispy, which do not include Kivy. The basic command for this would be e.g.:

```
python-for-android create --dist_name=testvispy --bootstrap=sdl2 --requirements=vispy
```

python-for-android also has commands to list internal information about distributions available, to export or symlink these (they come with a standalone APK build script), and in future will also support features including binary download to avoid the manual compilation step.

See the Commands documentation for full details of available functionality.

1.1.4 Setting paths to the the SDK and NDK

If building your own dists it is necessary to have installed the Android SDK and NDK, and to make Kivy aware of their locations. The instructions in *basic use* use environment variables for this, but this is not the only option. The different possibilities for each setting are given below.

Path to the Android SDK

python-for-android searches in the following places for this path, in order; setting any of these variables overrides all the later ones:

- The `--sdk_path` argument to any python-for-android command.
- The `ANDROIDSDK` environment variable.
- The `ANDROID_HOME` environment variable (this may be used or set by other tools).
- By using buildozer and letting it download the SDK; python-for-android automatically checks the default buildozer download directory. This is intended to make testing python-for-android easy.

If none of these is set, python-for-android will raise an error and exit.

The Android API to target

When building for Android it is necessary to target an API number corresponding to a specific version of Android. Whatever you choose, your APK will probably not work in earlier versions, but you also cannot use features introduced in later versions.

You must download specific platform tools for the SDK for any given target, it does not come with any. Do this by running `/path/to/android/sdk/tools/android`, which will give a gui interface, and select the 'platform tools' option under your chosen target.

The default target of python-for-android is 14, corresponding to Android 4.0. This may be changed in the near future.

You must pass the target API to python-for-android, and can do this in several ways. Each choice overrides all the later ones:

- The `--android_api` argument to any python-for-android command.
- The `ANDROIDAPI` environment variables.
- If neither of the above, the default target is used (currently 14).

python-for-android checks if the target you select is available, and gives an error if not, so it's easy to test if you passed this variable correctly.

Path to the Android NDK

python-for-android searches in the following places for this path, in order; setting any of these variables overrides all the later ones:

- The `--ndk_path` argument to any python-for-android command.
- The `ANDROIDNDK` environment variable.
- The `NDK_HOME` environment variable (this may be used or set by other tools).
- The `ANDROID_NDK_HOME` environment variable (this may be used or set
- By using buildozer and letting it download the NDK; python-for-android automatically checks the default buildozer download directory. This is intended to make testing python-for-android easy. by other tools).

If none of these is set, python-for-android will raise an error and exit.

The Android NDK version

python-for-android needs to know what version of the NDK is installed, in order to properly resolve its internal filepaths. You can set this with any of the following methods - note that the first is preferred, and means that you probably do *not* have to manually set this.

- The `RELEASE.TXT` file in the NDK directory. If this exists and contains the version (which it probably does automatically), you do not need to set it manually.
- The `--ndk_ver` argument to any python-for-android command.
- The `ANDROIDNDKVER` environment variable.

If `RELEASE.TXT` exists but you manually set a different version, python-for-android will warn you about it, but will assume you are correct and try to continue the build.

1.2 Commands

This page documents all the commands and options that can be passed to toolchain.py.

1.2.1 Commands index

The commands available are the methods of the ToolchainCL class, documented below. They may have options of their own, or you can always pass *general arguments* or *distribution arguments* to any command (though if irrelevant they may not have an effect).

1.2.2 General arguments

These arguments may be passed to any command in order to modify its behaviour, though not all commands make use of them.

--debug Print extra debug information about the build, including all compilation output.

--sdk_dir The filepath where the Android SDK is installed. This can alternatively be set in several other ways.

--android_api The Android API level to target; python-for-android will check if the platform tools for this level are installed.

--ndk_dir The filepath where the Android NDK is installed. This can alternatively be set in several other ways.

--ndk_version The version of the NDK installed, important because the internal filepaths to build tools depend on this. This can alternatively be set in several other ways, or if your NDK dir contains a RELEASE.TXT containing the version this is automatically checked so you don't need to manually set it.

1.2.3 Distribution arguments

p4a supports several arguments used for specifying which compiled Android distribution you want to use. You may pass any of these arguments to any command, and if a distribution is required they will be used to load, or compile, or download this as necessary.

None of these options are essential, and in principle you need only supply those that you need.

--name NAME The name of the distribution. Only one distribution with a given name can be created.

--requirements LIST,OF,REQUIREMENTS The recipes that your distribution must contain, as a comma separated list. These must be names of recipes or the pypi names of Python modules.

--force_build BOOL Whether the distribution must be compiled from scratch.

Note: These options are preliminary. Others will include toggles for allowing downloads, and setting additional directories from which to load user dists.

1.3 Recipes

This documentation describes how python-for-android (p4a) recipes work. These are special scripts for installing different programs (including Python modules) into a p4a distribution. They are necessary to take care of compilation for any compiled components, as these must be compiled for Android with the correct architecture.

python-for-android comes with many recipes for popular modules, and no recipe is necessary at all for the use of Python modules with no compiled components; if you just want to build an APK, you can jump straight to the Quick-start or Commands documentation, or can use the `recipes` command to list available recipes.

If you are new to building recipes, it is recommended that you first read all of this page, at least up to the Recipe reference documentation. The different recipe sections include a number of examples of how recipes are built or overridden for specific purposes.

1.3.1 Creating your own Recipe

This documentation jumps straight to the practicalities of creating your own recipe. The formal reference documentation of the Recipe class can be found in the *Recipe class* section and below.

Check the *recipe template section* for a template that combines all of these ideas, in which you can replace whichever components you like.

The basic declaration of a recipe is as follows:

```
class YourRecipe(Recipe):

    url = 'http://example.com/example-{version}.tar.gz'
    version = '2.0.3'
    md5sum = '4f3dc9a9d857734a488bcbefd9cd64ed'

    depends = ['kivy', 'sdl2']  # These are just examples
    conflicts = ['pygame']

recipe = YourRecipe()
```

See the *Recipe class documentation* for full information about each parameter.

These core options are vital for all recipes, though the url may be omitted if the source is somehow loaded from elsewhere.

The `recipe = YourRecipe()` is also vital. This variable is used when the recipe is imported as the recipe instance to build with. If it is omitted, your recipe won't work.

Note: The url includes the `{version}` tag. You should only access the url with the `versioned_url` property, which replaces this with the version attribute.

The actual build process takes place via three core methods:

```
def prebuild_arch(self, arch):
    super(YourRecipe, self).prebuild_arch(arch)
    # Do any pre-initialisation

def build_arch(self, arch):
    super(YourRecipe, self).build_arch(arch)
    # Do the main recipe build

def postbuild_arch(self, arch):
    super(YourRecipe, self).build_arch(arch)
    # Do any clearing up
```

The prebuild of every recipe is run before the build of any recipe, and likewise the build of every recipe is run before the postbuild of any. This lets you strictly order the build process.

If you defined an url for your recipe, you do *not* need to manually download it, this is handled automatically.

The recipe will automatically be built in a special isolated build directory, which you can access with `self.get_build_dir(arch.arch)`. You should only work within this directory. It may be convenient to use the `current_directory` context manager defined in toolchain.py:

```
from pythonforandroid.toolchain import current_directory
def build_arch(self, arch):
    super(YourRecipe, self).build_arch(arch)
    with current_directory(self.get_build_dir(arch.arch)):
        with open('example_file.txt', 'w'):
            fileh.write('This is written to a file within the build dir')
```

The argument to each method, `arch`, is an object relating to the architecture currently being built for. You can mostly ignore it, though may need to use the arch name `arch.arch`.

Note: You can also implement arch-specific versions of each method, which are called (if they exist) by the superclass, e.g. `def prebuild_armeabi(self, arch)`.

This is the core of what's necessary to write a recipe, but has not covered any of the details of how one actually writes code to compile for android. This is covered in the next sections, including the *standard mechanisms* used as part of the build, and the details of specific recipe classes for Python, Cython, and some generic compiled recipes. If your module is one of the latter, you should use these later classes rather than reimplementing the functionality from scratch.

1.3.2 Methods and tools to help with compilation

Patching modules before installation

You can easily apply patches to your recipes with the `apply_patch` method. For instance, you could do this in your prebuild method:

```
import sh
def prebuild_arch(self, arch):
     super(YourRecipe, self).prebuild_arch(arch)
     build_dir = self.get_build_dir(arch.arch)
     if exists(join(build_dir, '.patched')):
         print('Your recipe is already patched, skipping')
         return
     self.apply_patch('some_patch.patch')
     shprint(sh.touch, join(build_dir, '.patched'))
```

The path to the patch should be in relation to your recipe code. In this case, `some_path.patch` must be in the same directory as the recipe.

This code also manually takes care to patch only once. You can use the same strategy yourself, though a more generic solution may be provided in the future.

Installing libs

Some recipes generate .so files that must be manually copied into the android project. You can use code like the following to accomplish this, copying to the correct lib cache dir:

```
def build_arch(self, arch):
    do_the_build()  # e.g. running ./configure and make

    import shutil
    shutil.copyfile('a_generated_binary.so',
                    self.ctx.get_libs_dir(arch.arch))
```

Any libs copied to this dir will automatically be included in the appropriate libs dir of the generated android project.

Compiling for the Android architecture

When performing any compilation, it is vital to do so with appropriate environment variables set, ensuring that the Android libraries are properly linked and the compilation target is the correct architecture.

You can get a dictionary of appropriate environment variables with the `get_recipe_env` method. You should make sure to set this environment for any processes that you call. It is convenient to do this using the `sh` module as follows:

```
def build_arch(self, arch):
    super(YourRecipe, self).build_arch(arch)
    env = self.get_recipe_env(arch)
    sh.echo('$PATH', _env=env)  # Will print the PATH entry fron the
                                # env dict
```

You can also use the `shprint` helper function from the p4a toolchain module, which will print information about the process and its current status:

```
from pythonforandroid.toolchain import shprint
shprint(sh.echo, '$PATH', _env=env)
```

You can also override the `get_recipe_env` method to add new env vars for the use of your recipe. For instance, the Kivy recipe does the following when compiling for SDL2, in order to tell Kivy what backend to use:

```
def get_recipe_env(self, arch):
    env = super(KivySDL2Recipe, self).get_recipe_env(arch)
    env['USE_SDL2'] = '1'

    env['KIVY_SDL2_PATH'] = ':'.join([
        join(self.ctx.bootstrap.build_dir, 'jni', 'SDL', 'include'),
        join(self.ctx.bootstrap.build_dir, 'jni', 'SDL2_image'),
        join(self.ctx.bootstrap.build_dir, 'jni', 'SDL2_mixer'),
        join(self.ctx.bootstrap.build_dir, 'jni', 'SDL2_ttf'),
        ])
    return env
```

Warning: When using the sh module like this the new env *completely replaces* the normal environment, so you must define any env vars you want to access.

Including files with your recipe

The should_build method

The Recipe class has a `should_build` method, which returns a boolean. This is called before running `build_arch`, and if it returns False then the build is skipped. This is useful to avoid building a recipe more than once for different dists.

By default, should_build returns True, but you can override it however you like. For instance, PythonRecipe and its subclasses all replace it with a check for whether the recipe is already installed in the Python distribution:

```
def should_build(self):
    name = self.site_packages_name
    if name is None:
        name = self.name
    if exists(join(self.ctx.get_site_packages_dir(), name)):
```

```
        info('Python package already exists in site-packages')
        return False
    print('site packages', self.ctx.get_site_packages_dir())
    info('{} apparently isn\'t already in site-packages'.format(name))
    return True
```

1.3.3 Using a PythonRecipe

If your recipe is to install a Python module without compiled components, you should use a PythonRecipe. This overrides `build_arch` to automatically call the normal `python setup.py install` with an appropriate environment.

For instance, the following is all that's necessary to create a recipe for the Vispy module:

```
from pythonforandroid.toolchain import PythonRecipe
class VispyRecipe(PythonRecipe):
    version = 'master'
    url = 'https://github.com/vispy/vispy/archive/{version}.zip'

    depends = ['python2', 'numpy']

    site_packages_name = 'vispy'

recipe = VispyRecipe()
```

The `site_packages_name` is a new attribute that identifies the folder in which the module will be installed in the Python package. This is only essential to add if the name is different to the recipe name. It is used to check if the recipe installation can be skipped, which is the case if the folder is already present in the Python installation.

For reference, the code that accomplishes this is the following:

```
def build_arch(self, arch):
    super(PythonRecipe, self).build_arch(arch)
    self.install_python_package()

def install_python_package(self):
    '''Automate the installation of a Python package (or a cython
    package where the cython components are pre-built).'''
    arch = self.filtered_archs[0]
    env = self.get_recipe_env(arch)

    info('Installing {} into site-packages'.format(self.name))

    with current_directory(self.get_build_dir(arch.arch)):
        hostpython = sh.Command(self.ctx.hostpython)

        shprint(hostpython, 'setup.py', 'install', '-O2', _env=env)
```

This combines techniques and tools from the above documentation to create a generic mechanism for all Python modules.

Note: The hostpython is the path to the Python binary that should be used for any kind of installation. You *must* run Python in a similar way if you need to do so in any of your own recipes.

1.3.4 Using a CythonRecipe

If your recipe is to install a Python module that uses Cython, you should use a CythonRecipe. This overrides `build_arch` to both build the cython components and to install the Python module just like a normal PythonRecipe.

For instance, the following is all that's necessary to make a recipe for Kivy (in this case, depending on Pygame rather than SDL2):

```
class KivyRecipe(CythonRecipe):
     version = 'stable'
     url = 'https://github.com/kivy/kivy/archive/{version}.zip'
     name = 'kivy'

     depends = ['pygame', 'pyjnius', 'android']

recipe = KivyRecipe()
```

For reference, the code that accomplishes this is the following:

```
def build_arch(self, arch):
    Recipe.build_arch(self, arch)  # a hack to avoid calling
                                   # PythonRecipe.build_arch
    self.build_cython_components(arch)
    self.install_python_package()  # this is the same as in a PythonRecipe

def build_cython_components(self, arch):
    env = self.get_recipe_env(arch)
    with current_directory(self.get_build_dir(arch.arch)):
        hostpython = sh.Command(self.ctx.hostpython)

        # This first attempt *will* fail, because cython isn't
        # installed in the hostpython
        try:
            shprint(hostpython, 'setup.py', 'build_ext', _env=env)
        except sh.ErrorReturnCode_1:
            pass

        # ...so we manually run cython from the user's system
        shprint(sh.find, self.get_build_dir('armeabi'), '-iname', '*.pyx', '-exec',
                self.ctx.cython, '{}', ';', _env=env)

        # now cython has already been run so the build works
        shprint(hostpython, 'setup.py', 'build_ext', '-v', _env=env)

        # stripping debug symbols lowers the file size a lot
        build_lib = glob.glob('./build/lib*')
        shprint(sh.find, build_lib[0], '-name', '*.o', '-exec',
                env['STRIP'], '{}', ';', _env=env)
```

The failing build and manual cythonisation is necessary, first to make sure that any .pyx files have been generated by setup.py, and second because cython isn't installed in the hostpython build.

This may actually fail if the setup.py tries to import cython before making any pyx files (in which case it crashes too early), although this is probably not usually an issue. If this happens to you, try patching to remove this import or make it fail quietly.

Other than this, these methods follow the techniques in the above documentation to make a generic recipe for most cython based modules.

1.3.5 Using a CompiledComponentsPythonRecipe

This is similar to a CythonRecipe but is intended for modules like numpy which include compiled but non-cython components. It uses a similar mechanism to compile with the right environment.

This isn't documented yet because it will probably be changed so that CythonRecipe inherits from it (to avoid code duplication).

1.3.6 Using an NDKRecipe

If you are writing a recipe not for a Python module but for something that would normall go in the JNI dir of an Android project (i.e. it has an `Application.mk` and `Android.mk` that the Android build system can use), you can use an NDKRecipe to automatically set it up. The NDKRecipe overrides the normal `get_build_dir` method to place things in the Android project.

> **Warning:** The NDKRecipe does *not* currently actually call ndk-build, you must add this call (for your module) by manually making a build_arch method. This may be fixed later.

For instance, the following recipe is all that's necessary to place SDL2_ttf in the jni dir. This is built later by the SDL2 recipe, which calls ndk-build with this as a dependency:

```
class LibSDL2TTF(NDKRecipe):
    version = '2.0.12'
    url = 'https://www.libsdl.org/projects/SDL_ttf/release/SDL2_ttf-{version}.tar.gz'
    dir_name = 'SDL2_ttf'

recipe = LibSDL2TTF()
```

The dir_name argument is a new class attribute that tells the recipe what the jni dir folder name should be. If it is omitted, the recipe name is used. Be careful here, sometimes the folder name is important, especially if this folder is a dependency of something else.

1.3.7 A Recipe template

The following template includes all the recipe sections you might use. Note that none are compulsory, feel free to delete method overrides if you do not use them:

```
from pythonforandroid.toolchain import Recipe, shprint, current_directory
from os.path import exists, join
import sh
import glob


class YourRecipe(Recipe):
    # This could also inherit from PythonRecipe etc. if you want to
    # use their pre-written build processes

    version = 'some_version_string'
    url = 'http://example.com/example-{version}.tar.gz'

    depends = ['python2', 'numpy']  # A list of any other recipe names
                                    # that must be built before this
                                    # one

    conflicts = []  # A list of any recipe names that cannot be built
```

```
                    # alongside this one

    def get_recipe_env(self, arch):
        env = super(YourRecipe, self).get_recipe_env()
        # Manipulate the env here if you want
        return env

    def should_build(self):
        # Add a check for whether the recipe is already built if you
        # want, and return False if it is.
        return True

    def prebuild_arch(self, arch):
        super(YourRecipe, self).prebuild_arch(self)
        # Do any extra prebuilding you want, e.g.:
        self.apply_patch('path/to/patch.patch')

    def build_arch(self, arch):
        super(YourRecipe, self).build_arch(self)
        # Build the code. Make sure to use the right build dir, e.g.
        with current_directory(self.get_build_dir(arch.arch)):
            sh.ls('-lathr')  # Or run some commands that actually do
                             # something

    def postbuild_arch(self, arch):
        super(YourRecipe, self).prebuild_arch(self)
        # Do anything you want after the build, e.g. deleting
        # unnecessary files such as documentation


recipe = YourRecipe()
```

1.3.8 Examples of recipes

The above documentation has included a number of snippets demonstrating different behaviour. Together, these cover most of what is ever necessary to make a recipe work.

The following short sections further demonstrate a few full recipes from p4a's internal recipes folder. Unless your own module has some unusual complication, following these templates should be all you need to make your own recipes work.

TODO

1.3.9 The `Recipe` class

The `Recipe` is the base class for all p4a recipes. The core documentation of this class is given below, followed by discussion of how to create your own Recipe subclass.

1.4 Bootstraps

python-for-android (p4a) supports multiple *bootstraps*. These fulfil a similar role to recipes, but instead of describing how to compile a specific module they describe how a full Android project may be put together from a combination of individual recipes and other components such as Android source code and various build files.

If you do not want to modify p4a, you don't need to worry about bootstraps, just make sure you specify what modules you want to use (or specify an existing bootstrap manually), and p4a will automatically build everything appropriately.

This page describes the basics of how bootstraps work so that you can create and use your own if you like, making it easy to build new kinds of Python project for Android.

1.4.1 Current bootstraps

python-for-android includes the following bootstraps by default, which may be chosen by name with a build parameter, or (by default) are selected automatically in order to fulfil your build requirements. For instance, if you add 'sdl2' in the requirements, the sdl2 backend will be used.

p4a is designed to make it fairly easy to make your own bootstrap with a new backend, e.g. one that creates a webview interface and runs python in the background to serve a flask or django site from the phone itself.

pygame

This builds APKs exactly like the old p4a toolchain, using Pygame as the windowing and input backend.

This bootstrap automatically includes pygame, kivy, and python. It could potentially be modified to work for non-Kivy projects.

sdl2

This builds APKs using SDL2 as the window and input backend. It is not fully developed compared to the Pygame backend, but has many advantages and will be the long term default.

This bootstrap automatically includes SDL2, but nothing else.

You can use the sdl2 bootstrap to seamlessly make a Kivy APK, but can also make Python apps using other libraries; for instance, using pysdl2 and pyopengl. Vispy also runs on android this way.

empty

This bootstrap has no dependencies and cannot actually build an APK. It is useful for testing recipes without building unnecessary components.

1.4.2 Creating a new bootstrap

A bootstrap class consists of just a few basic components, though one of them must do a lot of work.

For instance, the SDL2 bootstrap looks like the following:

```
from pythonforandroid.toolchain import Bootstrap, shprint, current_directory, info, warning, ArchAndr
from os.path import join, exists
from os import walk
import glob
import sh


class SDL2Bootstrap(Bootstrap):
    name = 'sdl2'

    recipe_depends = ['sdl2']
```

```
    def run_distribute(self):
        # much work is done here...
```

The declaration of the bootstrap name and recipe dependencies should be clear. However, the `run_distribute` method must do all the work of creating a build directory, copying recipes etc into it, and adding or removing any extra components as necessary.

If you'd like to creat a bootstrap, the best resource is to check the existing ones in the p4a source code. You can also contact the developers if you have problems or questions.

1.5 Accessing Android APIs

When writing an Android application you may want to access the normal Android APIs, which are available in Java. It is by calling these that you would normally accomplish everything from vibration, to opening other applications, to accessing sensor data, to controlling settings like screen orientation and wakelocks.

These APIs can be accessed from Python to perform all of these tasks and many more. This is made possible by the Pyjnius module, a Python library for automatically wrapping Java and making it callable from Python code. This is fairly simple to use, though not very Pythonic and inherits Java's verbosity. For this reason the Kivy organisation also created Plyer, which further wraps specific APIs in a Pythonic and cross-platform way - so in fact you can call the same code in Python but have it do the right thing also on platforms other than Android.

These are both independent projects whose documentation is linked above, and you can check this to learn about all the things they can do. The following sections give some simple introductory examples, along with explanation of how to include these modules in your APKs.

1.5.1 Using Pyjnius

Pyjnius lets you call the Android API directly from Python; this let's you do almost everything you can (and probably would) do in a Java app. Pyjnius is works by dynamically wrapping Java classes, so you don't have to wait for any particular feature to be pre-supported.

You can include Pyjnius in your APKs by adding the *pyjnius* or *pyjniussdl2* recipes to your build requirements (the former works with Pygame/SDL1, the latter with SDL2, the need to make this choice will be removed later when pyjnius internally supports multiple Android backends). It is automatically included in any APK containing Kivy, in which case you don't need to specify it manually.

The basic mechanism of Pyjnius is the *autoclass* command, which wraps a Java class. For instance, here is the code to vibrate your device:

```
from jnius import autoclass

# We need a reference to the Java activity running the current
# application, this reference is stored automatically by
# Kivy's PythonActivity bootstrap

# This one works with Pygame
# PythonActivity = autoclass('org.renpy.android.PythonActivity')

# This one works with SDL2
PythonActivity = autoclass('org.kivy.android.PythonActivity')

activity = PythonActivity.mActivity

Context = autoclass('android.content.Context')
```

```
vibrator = activity.getSystemService(Context.VIBRATOR_SERVICE)

vibrator.vibrate(10000)  # the argument is in milliseconds
```

Things to note here are:

- The class that must be wrapped depends on the bootstrap. This is because Pyjnius is using the bootstrap's java source code to get a reference to the current activity, which both the Pygame and SDL2 bootstraps store in the `mActivity` static variable. This difference isn't always important, but it's important to know about.
- The code closely follows the Java API - this is exactly the same set of function calls that you'd use to achieve the same thing from Java code.
- This is quite verbose - it's a lot of lines to achieve a simple vibration!

These emphasise both the advantages and disadvantage of Pyjnius; you *can* achieve just about any API call with it (though the syntax is sometimes a little more involved, particularly if making Java classes from Python code), but it's not Pythonic and it's not short. These are problems that Plyer, explained below, attempts to address.

You can check the Pyjnius documentation for further details.

1.5.2 Using Plyer

Plyer aims to provide a much less verbose, Pythonic wrapper to platform-specific APIs. Android is a supported platform, but it also supports iOS and desktop operating systems, with the idea that the same Plyer code would do the right thing on any of them, though Plyer is a work in progress and not all platforms support all Plyer calls yet. This is the disadvantage of Plyer, it does not support all APIs yet, but you can always Pyjnius to call anything that is currently missing.

You can include Plyer in your APKs by adding the *Plyer* recipe to your build requirements. It is not included automatically.

You should check the Plyer documentation for details of all supported facades (platform APIs), but as an example the following is how you would achieve vibration as described in the Pyjnius section above:

```
from plyer.vibrator import vibrate
vibrate(10)  # in Plyer, the argument is in seconds
```

This is obviously *much* less verbose!

> **Warning:** At the time of writing, the Plyer recipe is not yet ported, and Plyer doesn't support SDL2. These issues will be fixed soon.

1.6 Troubleshooting

1.6.1 Debug output

Add the `--debug` option to any python-for-android command to see full debug output including the output of all the external tools used in the compilation and packaging steps.

If reporting a problem by email or irc, it is usually helpful to include this full log, via e.g. a pastebin or Github gist.

1.6.2 Getting help

python-for-android is managed by the Kivy Organisation, and you can get help with any problems using the same channels as Kivy itself:

- by email to the kivy-users Google group
- by irc in the #kivy room at irc.freenode.net

If you find a bug, you can also post an issue on the python-for-android Github page.

1.6.3 Debugging on Android

When a python-for-android APK doesn't work, often the only indication that you get is that it closes. It is important to be able to find out what went wrong.

python-for-android redirects Python's stdout and stderr to the Android logcat stream. You can see this by enabling developer mode on your Android device, enabling adb on the device, connecting it to your PC (you should see a notification that USB debugging is connected) and running `adb logcat`. If adb is not in your PATH, you can find it at `/path/to/Android/SDK/platform-tools/adb`, or access it through python-for-android with the shortcut:

```
python-for-android logcat
```

or:

```
python-for-android adb logcat
```

Running logcat command gives a lot of information about what Android is doing. You can usually see important lines by using logcat's built in functionality to see only lines with the `python` tag (or just grepping this).

When your app crashes, you'll see the normal Python traceback here, as well as the output of any print statements etc. that your app runs. Use these to diagnose the problem just as normal.

The adb command passes its arguments straight to adb itself, so you can also do other debugging tasks such as `python-for-android adb devices` to get the list of connected devices.

For further information, see the Android docs on adb, and on logcat in particular.

1.6.4 Common errors

The following are common errors that can arise during the use of python-for-android, along with solutions where possible.

1.7 Differences to the old python-for-android project

This python-for-android project is a rewrite, and eventual replacement, for Kivy's existing python-for-android project. This project existed for some time and, although it is now close to deprecated, there are many resources discussing how to use it.

For this reason, the following documentation gives a basic explanation of how to accomplish that previously used the `distribute.sh` and `build.py` toolchain with this new one.

1.8 Contributing

The development of python-for-android is managed by the Kivy team via Github.

Issues and pull requests are welcome via the integrated issue tracker.

1.9 Old p4a toolchain doc

This is the documentation for the old python-for-android toolchain, using distribute.sh and build.py. This it entirely superseded by the new toolchain, you do not need to read it unless using this old method.

Python for android is a project to create your own Python distribution including the modules you want, and create an apk including python, libs, and your application.

- Forum: https://groups.google.com/forum/#!forum/python-android
- Mailing list: python-android@googlegroups.com

1.9.1 Toolchain

Introduction

In terms of comparaison, you can check how Python for android can be useful compared to other projects.

Project	Native Python	GUI libraries	APK generation	Custom build
Python for android	Yes	Yes	Yes	Yes
PGS4A	Yes	Yes	Yes	No
Android scripting	No	No	No	No
Python on a chip	No	No	No	No

Note: For the moment, we are shipping only one "java bootstrap" (needed for decompressing your packaged zip file project, create an OpenGL ES 2.0 surface, handle touch input and manage an audio thread).

If you want to use it without kivy module (an opengl es 2.0 ui toolkit), then you might want a lighter java bootstrap, that we don't have right now. Help is welcome :)

So for the moment, Python for Android can only be used with the kivy GUI toolkit: http://kivy.org/#home

How does it work ?

To be able to run Python on android, you need to compile it for android. And you need to compile all the libraries you want for android too. Since Python is a language, not a toolkit, you cannot draw any user interface with it: you need to use a toolkit for it. Kivy can be one of them.

So for a simple ui project, the first step is to compile Python + Kivy + all others libraries. Then you'll have what we call a "distribution". A distribution is composed of:

- Python
- Python libraries
- All selected libraries (kivy, pygame, pil...)
- A java bootstrap
- A build script

You'll use the build script for create an "apk": an android package.

Prerequisites

Note: There is a VirtualBox Image we provide with the prerequisites along with the Android SDK and NDK preinstalled to ease your installation woes. You can download it from here.

Warning: The current version is tested only on Ubuntu oneiric (11.10) and precise (12.04). If it doesn't work on other platforms, send us a patch, not a bug report. Python for Android works on Linux and Mac OS X, not Windows.

You need the minimal environment for building python. Note that other libraries might need other tools (cython is used by some recipes, and ccache to speedup the build):

```
sudo apt-get install build-essential patch git-core ccache ant python-pip python-dev
```

If you are on a 64 bit distro, you should install these packages too :

```
sudo apt-get install ia32-libs  libc6-dev-i386
```

On debian Squeeze amd64, those packages were found to be necessary :

```
sudo apt-get install lib32stdc++6 lib32z1
```

Ensure you have the latest Cython version:

```
pip install --upgrade cython
```

You must have android SDK and NDK. The SDK defines the Android functions you can use. The NDK is used for compilation. Right now, it's preferred to use:

- SDK API 8 or 14 (15 will only work with a newly released NDK)
- NDK r5b or r7

You can download them at:

```
http://developer.android.com/sdk/index.html
http://developer.android.com/sdk/ndk/index.html
```

In general, Python for Android currently works with Android 2.3 to L.

If it's your very first time using the Android SDK, don't forget to follow the documentation for recommended components at:

```
http://developer.android.com/sdk/installing/adding-packages.html

    You need to download at least one platform into your environment, so
    that you will be able to compile your application and set up an Android
    Virtual Device (AVD) to run it on (in the emulator). To start with,
    just download the latest version of the platform. Later, if you plan to
    publish your application, you will want to download other platforms as
    well, so that you can test your application on the full range of
    Android platform versions that your application supports.
```

After installing them, export both installation paths, NDK version, and API to use:

```
export ANDROIDSDK=/path/to/android-sdk
export ANDROIDNDK=/path/to/android-ndk
export ANDROIDNDKVER=rX
export ANDROIDAPI=X

# example
export ANDROIDSDK="/home/tito/code/android/android-sdk-linux_86"
export ANDROIDNDK="/home/tito/code/android/android-ndk-r7"
export ANDROIDNDKVER=r7
export ANDROIDAPI=14
```

Also, you must configure your PATH to add the `android` binary:

```
export PATH=$ANDROIDNDK:$ANDROIDSDK/platform-tools:$ANDROIDSDK/tools:$PATH
```

Usage

Step 1: compile the toolchain

If you want to compile the toolchain with only the kivy module:

```
./distribute.sh -m "kivy"
```

Warning: Do not run the above command from within a virtual enviroment.

After a long time, you'll get a "dist/default" directory containing all the compiled libraries and a build.py script to package your application using thoses libraries.

You can include other modules (or "recipes") to compile using *-m*:

```
./distribute.sh -m "openssl kivy"
./distribute.sh -m "pil ffmpeg kivy"
```

Note: Recipes are instructions for compiling Python modules that require C extensions. The list of recipes we currently have is at: https://github.com/kivy/python-for-android/tree/master/recipes

You can also specify a specific version for each package. Please note that the compilation might **break** if you don't use the default version. Most recipes have patches to fix Android issues, and might not apply if you specify a version. We also recommend to clean build before changing version.:

```
./distribute.sh -m "openssl kivy==master"
```

Python modules that don't need C extensions don't need a recipe and can be included this way. From python-for-android 1.1 on, you can now specify pure-python package into the distribution. It will use virtualenv and pip to install pure-python modules into the distribution. Please note that the compiler is deactivated, and will break any module which tries to compile something. If compilation is needed, write a recipe:

```
./distribute.sh -m "requests pygments kivy"
```

Note: Recipes download a defined version of their needed package from the internet, and build from it. If you know what you are doing, and want to override that, you can export the env variable *P4A_recipe_name_DIR* and this directory will be copied and used instead.

Available options to *distribute.sh*:

```
-d directory           Name of the distribution directory
-h                     Show this help
-l                     Show a list of available modules
-m 'mod1 mod2'         Modules to include
-f                     Restart from scratch (remove the current build)
-u 'mod1 mod2'         Modules to update (if already compiled)
```

Step 2: package your application

Go to your custom Python distribution:

```
cd dist/default
```

Use the build.py for creating the APK:

```
./build.py --package org.test.touchtracer --name touchtracer \
--version 1.0 --dir ~/code/kivy/examples/demo/touchtracer debug
```

Then, the Android package (APK) will be generated at:

bin/touchtracer-1.0-debug.apk

Warning: Some files and modules for python are blacklisted by default to save a few megabytes on the final APK file. In case your applications doesn't find a standard python module, check the src/blacklist.txt file, remove the module you need from the list, and try again.

Available options to *build.py*:

```
-h, --help            show this help message and exit
--package PACKAGE     The name of the java package the project will be
                      packaged under.
--name NAME           The human-readable name of the project.
--version VERSION     The version number of the project. This should consist
                      of numbers and dots, and should have the same number
                      of groups of numbers as previous versions.
--numeric-version NUMERIC_VERSION
                      The numeric version number of the project. If not
                      given, this is automatically computed from the
                      version.
--dir DIR             The directory containing public files for the project.
--private PRIVATE     The directory containing additional private files for
                      the project.
--launcher            Provide this argument to build a multi-app launcher,
                      rather than a single app.
--icon-name ICON_NAME
                      The name of the project's launcher icon.
--orientation ORIENTATION
                      The orientation that the game will display in. Usually
                      one of "landscape", "portrait" or "sensor".
--permission PERMISSIONS
                      The permissions to give this app.
--ignore-path IGNORE_PATH
                      Ignore path when building the app
--icon ICON           A png file to use as the icon for the application.
--presplash PRESPLASH
                      A jpeg file to use as a screen while the application
                      is loading.
```

```
--install-location INSTALL_LOCATION
                      The default install location. Should be "auto",
                      "preferExternal" or "internalOnly".
--compile-pyo         Compile all .py files to .pyo, and only distribute the
                      compiled bytecode.
--intent-filters INTENT_FILTERS
                      Add intent-filters xml rules to AndroidManifest.xml
--blacklist BLACKLIST
                      Use a blacklist file to match unwanted file in the
                      final APK
--sdk SDK_VERSION     Android SDK version to use. Default to 8
--minsdk MIN_SDK_VERSION
                      Minimum Android SDK version to use. Default to 8
--window              Indicate if the application will be windowed
```

Meta-data

New in version 1.3.

You can extend the *AndroidManifest.xml* with application meta-data. If you are using external toolkits like Google Maps, you might want to set your API key in the meta-data. You could do it like this:

```
./build.py ... --meta-data com.google.android.maps.v2.API_KEY=YOURAPIKEY
```

Some meta-data can be used to interact with the behavior of our internal component.

Token	Description
surface.transparent	If set to 1, the created surface will be transparent (can be used to add background Android widget in the background, or use accelerated widgets)
surface.depth	Size of the depth component, default to 0. 0 means automatic, but you can force it to a specific value. Be warned, some old phone might not support the depth you want.
surface.stencil	Size of the stencil component, default to 8.
android.background_color	Color (32bits RGBA color), used for the background window. Usually, the background is covered by the OpenGL Background, unless *surface.transparent* is set.

Customize your distribution

The basic layout of a distribution is:

```
AndroidManifest.xml     - (*) android manifest (generated from templates)
assets/
    private.mp3         - (*) fake package that will contain all the python installation
    public.mp3          - (*) fake package that will contain your application
bin/                    - contain all the apk generated from build.py
blacklist.txt           - list of file patterns to not include in the APK
buildlib/               - internals libraries for build.py
build.py                - build script to use for packaging your application
build.xml               - (*) build settings (generated from templates)
default.properties      - settings generated from your distribute.sh
libs/                   - contain all the compiled libraries
local.properties        - settings generated from your distribute.sh
private/                - private directory containing all the python files
    lib/                  this is where you can remove or add python libs.
        python2.7/        by default, some modules are already removed (tests, idlelib, ...)
project.properties      - settings generated from your distribute.sh
python-install/         - the whole python installation, generated from distribute.sh
```

```
                          not included in the final package.
res/                    - (*) android resource (generated from build.py)
src/                    - Java bootstrap
templates/              - Templates used by build.py

(*): Theses files are automatically generated from build.py, don't change them directly !
```

Prerequisites

Note: There is a VirtualBox Image we provide with the prerequisites along with the Android SDK and NDK preinstalled to ease your installation woes. You can download it from here.

Warning: The current version is tested only on Ubuntu oneiric (11.10) and precise (12.04). If it doesn't work on other platforms, send us a patch, not a bug report. Python for Android works on Linux and Mac OS X, not Windows.

You need the minimal environment for building python. Note that other libraries might need other tools (cython is used by some recipes, and ccache to speedup the build):

```
sudo apt-get install build-essential patch git-core ccache ant python-pip python-dev
```

If you are on a 64 bit distro, you should install these packages too :

```
sudo apt-get install ia32-libs  libc6-dev-i386
```

On debian Squeeze amd64, those packages were found to be necessary :

```
sudo apt-get install lib32stdc++6 lib32z1
```

Ensure you have the latest Cython version:

```
pip install --upgrade cython
```

You must have android SDK and NDK. The SDK defines the Android functions you can use. The NDK is used for compilation. Right now, it's preferred to use:

- SDK API 8 or 14 (15 will only work with a newly released NDK)
- NDK r5b or r7

You can download them at:

```
http://developer.android.com/sdk/index.html
http://developer.android.com/sdk/ndk/index.html
```

In general, Python for Android currently works with Android 2.3 to L.

If it's your very first time using the Android SDK, don't forget to follow the documentation for recommended components at:

```
http://developer.android.com/sdk/installing/adding-packages.html

    You need to download at least one platform into your environment, so
    that you will be able to compile your application and set up an Android
    Virtual Device (AVD) to run it on (in the emulator). To start with,
    just download the latest version of the platform. Later, if you plan to
    publish your application, you will want to download other platforms as
```

```
    well, so that you can test your application on the full range of
    Android platform versions that your application supports.
```

After installing them, export both installation paths, NDK version, and API to use:

```
export ANDROIDSDK=/path/to/android-sdk
export ANDROIDNDK=/path/to/android-ndk
export ANDROIDNDKVER=rX
export ANDROIDAPI=X

# example
export ANDROIDSDK="/home/tito/code/android/android-sdk-linux_86"
export ANDROIDNDK="/home/tito/code/android/android-ndk-r7"
export ANDROIDNDKVER=r7
export ANDROIDAPI=14
```

Also, you must configure your PATH to add the `android` binary:

```
export PATH=$ANDROIDNDK:$ANDROIDSDK/platform-tools:$ANDROIDSDK/tools:$PATH
```

Usage

Step 1: compile the toolchain If you want to compile the toolchain with only the kivy module:

```
./distribute.sh -m "kivy"
```

Warning: Do not run the above command from within a virtual enviroment.

After a long time, you'll get a "dist/default" directory containing all the compiled libraries and a build.py script to package your application using thoses libraries.

You can include other modules (or "recipes") to compile using *-m*:

```
./distribute.sh -m "openssl kivy"
./distribute.sh -m "pil ffmpeg kivy"
```

Note: Recipes are instructions for compiling Python modules that require C extensions. The list of recipes we currently have is at: https://github.com/kivy/python-for-android/tree/master/recipes

You can also specify a specific version for each package. Please note that the compilation might **break** if you don't use the default version. Most recipes have patches to fix Android issues, and might not apply if you specify a version. We also recommend to clean build before changing version.:

```
./distribute.sh -m "openssl kivy==master"
```

Python modules that don't need C extensions don't need a recipe and can be included this way. From python-for-android 1.1 on, you can now specify pure-python package into the distribution. It will use virtualenv and pip to install pure-python modules into the distribution. Please note that the compiler is deactivated, and will break any module which tries to compile something. If compilation is needed, write a recipe:

```
./distribute.sh -m "requests pygments kivy"
```

Note: Recipes download a defined version of their needed package from the internet, and build from it. If you know what you are doing, and want to override that, you can export the env variable *P4A_recipe_name_DIR* and this directory will be copied and used instead.

Available options to *distribute.sh*:

```
-d directory            Name of the distribution directory
-h                      Show this help
-l                      Show a list of available modules
-m 'mod1 mod2'          Modules to include
-f                      Restart from scratch (remove the current build)
-u 'mod1 mod2'          Modules to update (if already compiled)
```

Step 2: package your application Go to your custom Python distribution:

```
cd dist/default
```

Use the build.py for creating the APK:

```
./build.py --package org.test.touchtracer --name touchtracer \
--version 1.0 --dir ~/code/kivy/examples/demo/touchtracer debug
```

Then, the Android package (APK) will be generated at:

bin/touchtracer-1.0-debug.apk

Warning: Some files and modules for python are blacklisted by default to save a few megabytes on the final APK file. In case your applications doesn't find a standard python module, check the src/blacklist.txt file, remove the module you need from the list, and try again.

Available options to *build.py*:

```
-h, --help            show this help message and exit
--package PACKAGE     The name of the java package the project will be
                      packaged under.
--name NAME           The human-readable name of the project.
--version VERSION     The version number of the project. This should consist
                      of numbers and dots, and should have the same number
                      of groups of numbers as previous versions.
--numeric-version NUMERIC_VERSION
                      The numeric version number of the project. If not
                      given, this is automatically computed from the
                      version.
--dir DIR             The directory containing public files for the project.
--private PRIVATE     The directory containing additional private files for
                      the project.
--launcher            Provide this argument to build a multi-app launcher,
                      rather than a single app.
--icon-name ICON_NAME
                      The name of the project's launcher icon.
--orientation ORIENTATION
                      The orientation that the game will display in. Usually
                      one of "landscape", "portrait" or "sensor".
--permission PERMISSIONS
                      The permissions to give this app.
--ignore-path IGNORE_PATH
                      Ignore path when building the app
--icon ICON           A png file to use as the icon for the application.
--presplash PRESPLASH
                      A jpeg file to use as a screen while the application
                      is loading.
--install-location INSTALL_LOCATION
```

```
                      The default install location. Should be "auto",
                      "preferExternal" or "internalOnly".
--compile-pyo         Compile all .py files to .pyo, and only distribute the
                      compiled bytecode.
--intent-filters INTENT_FILTERS
                      Add intent-filters xml rules to AndroidManifest.xml
--blacklist BLACKLIST
                      Use a blacklist file to match unwanted file in the
                      final APK
--sdk SDK_VERSION     Android SDK version to use. Default to 8
--minsdk MIN_SDK_VERSION
                      Minimum Android SDK version to use. Default to 8
--window              Indicate if the application will be windowed
```

Meta-data

New in version 1.3.

You can extend the *AndroidManifest.xml* with application meta-data. If you are using external toolkits like Google Maps, you might want to set your API key in the meta-data. You could do it like this:

```
./build.py ... --meta-data com.google.android.maps.v2.API_KEY=YOURAPIKEY
```

Some meta-data can be used to interact with the behavior of our internal component.

Token	Description
surface.transparent	If set to 1, the created surface will be transparent (can be used to add background Android widget in the background, or use accelerated widgets)
surface.depth	Size of the depth component, default to 0. 0 means automatic, but you can force it to a specific value. Be warned, some old phone might not support the depth you want.
surface.stencil	Size of the stencil component, default to 8.
android.background_color	Color (32bits RGBA color), used for the background window. Usually, the background is covered by the OpenGL Background, unless *surface.transparent* is set.

Customize your distribution

The basic layout of a distribution is:

```
AndroidManifest.xml     - (*) android manifest (generated from templates)
assets/
    private.mp3         - (*) fake package that will contain all the python installation
    public.mp3          - (*) fake package that will contain your application
bin/                    - contain all the apk generated from build.py
blacklist.txt           - list of file patterns to not include in the APK
buildlib/               - internals libraries for build.py
build.py                - build script to use for packaging your application
build.xml               - (*) build settings (generated from templates)
default.properties      - settings generated from your distribute.sh
libs/                   - contain all the compiled libraries
local.properties        - settings generated from your distribute.sh
private/                - private directory containing all the python files
    lib/                  this is where you can remove or add python libs.
        python2.7/        by default, some modules are already removed (tests, idlelib, ...)
project.properties      - settings generated from your distribute.sh
python-install/         - the whole python installation, generated from distribute.sh
```

```
                          not included in the final package.
res/                    - (*) android resource (generated from build.py)
src/                    - Java bootstrap
templates/              - Templates used by build.py

(*): Theses files are automatically generated from build.py, don't change them directly !
```

1.9.2 Examples

Prebuilt VirtualBox

A good starting point to build an APK are prebuilt VirtualBox images, where the Android NDK, the Android SDK, and the Kivy Python-For-Android sources are prebuilt in an VirtualBox image. Please search the Download Section for such an image. You will also need to create a device filter for the Android USB device using the VirtualBox OS settings.

Hello world

If you don't know how to start with Python for Android, here is a simple tutorial for creating an UI using Kivy, and make an APK with this project.

Note: Don't forget that Python for Android is not Kivy only, and you might want to use other toolkit libraries. When other toolkits will be available, this documentation will be enhanced.

Let's create a simple Hello world application, with one Label and one Button.

1. Ensure you've correctly installed and configured the project as said in the Prerequisites
2. Create a directory named `helloworld`:

```
mkdir helloworld
cd helloworld
```

3. Create a file named `main.py`, with this content:

```
import kivy
kivy.require('1.0.9')
from kivy.lang import Builder
from kivy.uix.gridlayout import GridLayout
from kivy.properties import NumericProperty
from kivy.app import App

Builder.load_string('''
<HelloWorldScreen>:
    cols: 1
    Label:
        text: 'Welcome to the Hello world'
    Button:
        text: 'Click me! %d' % root.counter
        on_release: root.my_callback()
''')

class HelloWorldScreen(GridLayout):
    counter = NumericProperty(0)
    def my_callback(self):
```

```
            print 'The button has been pushed'
            self.counter += 1

    class HelloWorldApp(App):
        def build(self):
            return HelloWorldScreen()

    if __name__ == '__main__':
        HelloWorldApp().run()
```

4. Go to the `python-for-android` directory
5. Create a distribution with kivy:

```
./distribute.sh -m kivy
```

6. Go to the newly created `default` distribution:

```
cd dist/default
```

7. Plug your android device, and ensure you can install development application
8. Build your hello world application in debug mode:

```
./build.py --package org.hello.world --name "Hello world" \
--version 1.0 --dir /PATH/TO/helloworld debug installd
```

9. Take your device, and start the application!
10. If something goes wrong, open the logcat by doing:

```
adb logcat
```

The final debug APK will be located in `bin/hello-world-1.0-debug.apk`.

If you want to release your application instead of just making a debug APK, you must:

1. Generate a non-signed APK:

```
./build.py --package org.hello.world --name "Hello world" \
--version 1.0 --dir /PATH/TO/helloworld release
```

2. Continue by reading http://developer.android.com/guide/publishing/app-signing.html

See also:

Kivy demos You can use them for creating APK too.

Compass

The following example is an extract from the Compass app as provided in the Kivy examples/android/compass folder:

```
# ... imports
Hardware = autoclass('org.renpy.android.Hardware')

class CompassApp(App):

    needle_angle = NumericProperty(0)

    def build(self):
        self._anim = None
```

```
        Hardware.magneticFieldSensorEnable(True)
        Clock.schedule_interval(self.update_compass, 1 / 10.)

    def update_compass(self, *args):
        # read the magnetic sensor from the Hardware class
        (x, y, z) = Hardware.magneticFieldSensorReading()

        # calculate the angle
        needle_angle = Vector(x , y).angle((0, 1)) + 90.

        # animate the needle
        if self._anim:
            self._anim.stop(self)
        self._anim = Animation(needle_angle=needle_angle, d=.2, t='out_quad')
        self._anim.start(self)

    def on_pause(self):
        # when you are going on pause, don't forget to stop the sensor
        Hardware.magneticFieldSensorEnable(False)
        return True

    def on_resume(self):
        # reactivate the sensor when you are back to the app
        Hardware.magneticFieldSensorEnable(True)

if __name__ == '__main__':
    CompassApp().run()
```

If you compile this app, you will get an APK which outputs the following screen:

Fig. 1.1: Screenshot of the Kivy Compass App (Source of the Compass Windrose: Wikipedia)

Hello world

If you don't know how to start with Python for Android, here is a simple tutorial for creating an UI using Kivy, and make an APK with this project.

Note: Don't forget that Python for Android is not Kivy only, and you might want to use other toolkit libraries. When other toolkits will be available, this documentation will be enhanced.

Let's create a simple Hello world application, with one Label and one Button.

1. Ensure you've correctly installed and configured the project as said in the Prerequisites
2. Create a directory named `helloworld`:

```
mkdir helloworld
cd helloworld
```

3. Create a file named `main.py`, with this content:

```
import kivy
kivy.require('1.0.9')
from kivy.lang import Builder
from kivy.uix.gridlayout import GridLayout
from kivy.properties import NumericProperty
from kivy.app import App

Builder.load_string('''
<HelloWorldScreen>:
    cols: 1
    Label:
        text: 'Welcome to the Hello world'
    Button:
        text: 'Click me! %d' % root.counter
        on_release: root.my_callback()
''')

class HelloWorldScreen(GridLayout):
    counter = NumericProperty(0)
    def my_callback(self):
        print 'The button has been pushed'
        self.counter += 1

class HelloWorldApp(App):
    def build(self):
        return HelloWorldScreen()

if __name__ == '__main__':
    HelloWorldApp().run()
```

4. Go to the `python-for-android` directory
5. Create a distribution with kivy:

```
./distribute.sh -m kivy
```

6. Go to the newly created `default` distribution:

```
cd dist/default
```

7. Plug your android device, and ensure you can install development application
8. Build your hello world application in debug mode:

```
./build.py --package org.hello.world --name "Hello world" \
--version 1.0 --dir /PATH/TO/helloworld debug installd
```

9. Take your device, and start the application!
10. If something goes wrong, open the logcat by doing:

```
adb logcat
```

The final debug APK will be located in `bin/hello-world-1.0-debug.apk`.

If you want to release your application instead of just making a debug APK, you must:

1. Generate a non-signed APK:

```
./build.py --package org.hello.world --name "Hello world" \
--version 1.0 --dir /PATH/TO/helloworld release
```

2. Continue by reading http://developer.android.com/guide/publishing/app-signing.html

See also:

Kivy demos You can use them for creating APK too.

Compass

The following example is an extract from the Compass app as provided in the Kivy examples/android/compass folder:

```
# ... imports
Hardware = autoclass('org.renpy.android.Hardware')

class CompassApp(App):

    needle_angle = NumericProperty(0)

    def build(self):
        self._anim = None
        Hardware.magneticFieldSensorEnable(True)
        Clock.schedule_interval(self.update_compass, 1 / 10.)

    def update_compass(self, *args):
        # read the magnetic sensor from the Hardware class
        (x, y, z) = Hardware.magneticFieldSensorReading()

        # calculate the angle
        needle_angle = Vector(x , y).angle((0, 1)) + 90.

        # animate the needle
        if self._anim:
            self._anim.stop(self)
        self._anim = Animation(needle_angle=needle_angle, d=.2, t='out_quad')
        self._anim.start(self)

    def on_pause(self):
        # when you are going on pause, don't forget to stop the sensor
        Hardware.magneticFieldSensorEnable(False)
        return True

    def on_resume(self):
        # reactivate the sensor when you are back to the app
        Hardware.magneticFieldSensorEnable(True)

if __name__ == '__main__':
    CompassApp().run()
```

If you compile this app, you will get an APK which outputs the following screen:

Fig. 1.2: Screenshot of the Kivy Compass App (Source of the Compass Windrose: Wikipedia)

1.9.3 Python API

The Python for Android project includes a Python module called `android` which consists of multiple parts that are mostly there to facilitate the use of the Java API.

This module is not designed to be comprehensive. Most of the Java API is also accessible with PyJNIus, so if you can't find what you need here you can try using the Java API directly instead.

Android (`android`)

`android.`**`check_pause`**`()`
: This should be called on a regular basis to check to see if Android expects the application to pause. If it returns true, the app should call *`android.wait_for_resume()`*, after storing its state as necessary.

`android.`**`wait_for_resume`**`()`
: This function should be called after *`android.check_pause()`* and returns true. It does not return until Android has resumed from the pause. While in this function, Android may kill the app without further notice.

`android.`**`map_key`**`(`*keycode*, *keysym*`)`
: This maps an android keycode to a python keysym. The android keycodes are available as constants in the android module.

Activity (`android.activity`)

The default PythonActivity has a observer pattern for onActivityResult and onNewIntent.

`android.activity.`**`bind`**`(`*eventname=callback*, ...`)`
: This allows you to bind a callback to an Android event: - `on_new_intent` is the event associated to the onNewIntent java call - `on_activity_result` is the event associated to the onActivityResult java call

> **Warning:** This method is not thread-safe. Call it in the mainthread of your app. (tips: use kivy.clock.mainthread decorator)

`android.activity.`**`unbind`**`(`*eventname=callback*, ...`)`
: Unregister a previously registered callback with *`bind()`*.

Example:

```
# This example is a snippet from an NFC p2p app implemented with Kivy.

from android import activity

def on_new_intent(self, intent):
    if intent.getAction() != NfcAdapter.ACTION_NDEF_DISCOVERED:
        return
    rawmsgs = intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES)
    if not rawmsgs:
        return
    for message in rawmsgs:
        message = cast(NdefMessage, message)
        payload = message.getRecords()[0].getPayload()
        print 'payload: {}'.format(''.join(map(chr, payload)))

def nfc_enable(self):
    activity.bind(on_new_intent=self.on_new_intent)
    # ...

def nfc_disable(self):
    activity.unbind(on_new_intent=self.on_new_intent)
    # ...
```

Billing (`android.billing`)

This billing module gives an access to the In-App Billing:

1. Setup a test account, and get your Public Key
2. Export your public key:

```
export BILLING_PUBKEY="Your public key here"
```

3. Setup some In-App product to buy. Let's say you've created a product with the id "org.kivy.gopremium"
4. In your application, you can use the `billing` module like this:

```
from android.billing import BillingService
from kivy.clock import Clock

class MyBillingService(object):

    def __init__(self):
        super(MyBillingService, self).__init__()

        # Start the billing service, and attach our callback
        self.service = BillingService(billing_callback)

        # Start a clock to check billing service message every second
        Clock.schedule_interval(self.service.check, 1)

    def billing_callback(self, action, *largs):
        '''Callback that will receive all the events from the Billing service
        '''
        if action == BillingService.BILLING_ACTION_ITEMSCHANGED:
            items = largs[0]
            if 'org.kivy.gopremium' in items:
```

```
                print "Congratulations, you have a premium acess"
            else:
                print "Unfortunately, you don't have premium access"

    def buy(self, sku):
        # Method to buy something.
        self.service.buy(sku)

    def get_purchased_items(self):
        # Return all the items purchased
        return self.service.get_purchased_items()
```

5. To initiate an in-app purchase, just call the `buy()` method:

```
# Note: start the service at the start, and never twice!
bs = MyBillingService()
bs.buy('org.kivy.gopremium')

# Later, when you get the notification that items have been changed, you
# can still check all the items you bought:
print bs.get_purchased_items()
{'org.kivy.gopremium': {'qt: 1}}
```

6. You'll receive all the notifications about the billing process in the callback.
7. Last step, create your application with `--with-billing $BILLING_PUBKEY`:

```
./build.py ... --with-billing $BILLING_PUBKEY
```

Broadcast (`android.broadcast`)

Implementation of the android BroadcastReceiver. You can specify the callback that will receive the broadcast event, and actions or categories filters.

class `android.broadcast.`**`BroadcastReceiver`**

> **Warning:** The callback will be called in another thread than the main thread. In that thread, be careful not to access OpenGL or something like that.

`__init__`(*callback*, *actions=None*, *categories=None*)

Parameters

- **`callback`** – function or method that will receive the event. Will receive the context and intent as argument.
- **`actions`** – list of strings that represent an action.
- **`categories`** – list of strings that represent a category.

For actions and categories, the string must be in lower case, without the prefix:

```
# In java: Intent.ACTION_HEADSET_PLUG
# In python: 'headset_plug'
```

`start`()
Register the receiver with all the actions and categories, and start handling events.

`stop`()
Unregister the receiver with all the actions and categories, and stop handling events.

Example:

```
class TestApp(App):

    def build(self):
        self.br = BroadcastReceiver(
            self.on_broadcast, actions=['headset_plug'])
        self.br.start()
        # ...

    def on_broadcast(self, context, intent):
        extras = intent.getExtras()
        headset_state = bool(extras.get('state'))
        if headset_state:
            print 'The headset is plugged'
        else:
            print 'The headset is unplugged'

    # Don't forget to stop and restart the receiver when the app is going
    # to pause / resume mode

    def on_pause(self):
        self.br.stop()
        return True

    def on_resume(self):
        self.br.start()
```

Mixer (`android.mixer`)

The *android.mixer* module contains a subset of the functionality in found in the pygame.mixer module. It's intended to be imported as an alternative to pygame.mixer, using code like:

```
try:
    import pygame.mixer as mixer
except ImportError:
    import android.mixer as mixer
```

Note that if you're using the kivy.core.audio module, you don't have to do anything, it is all automatic.

The *android.mixer* module is a wrapper around the Android MediaPlayer class. This allows it to take advantage of any hardware acceleration present, and also eliminates the need to ship codecs as part of an application.

It has several differences with the pygame mixer:

- The init() and pre_init() methods work, but are ignored - Android chooses appropriate settings automatically.
- Only filenames and true file objects can be used - file-like objects will probably not work.
- Fadeout does not work - it causes a stop to occur.
- Looping is all or nothing, there is no way to choose the number of loops that occur. For looping to work, the `android.mixer.periodic()` function should be called on a regular basis.
- Volume control is ignored.
- End events are not implemented.
- The mixer.music object is a class (with static methods on it), rather than a module. Calling methods like `mixer.music.play()` should work.

Runnable (android.runnable)

`Runnable` is a wrapper around the Java Runnable class. This class can be used to schedule a call of a Python function into the *PythonActivity* thread.

Example:

```
from android.runnable import Runnable

def helloworld(arg):
    print 'Called from PythonActivity with arg:', arg

Runnable(helloworld)('hello')
```

Or use our decorator:

```
from android.runnable import run_on_ui_thread

@run_on_ui_thread
def helloworld(arg):
    print 'Called from PythonActivity with arg:', arg

helloworld('arg1')
```

This can be used to prevent errors like:

- W/System.err(9514): java.lang.RuntimeException: Can't create handler inside thread that has not called Looper.prepare()
- NullPointerException in ActivityThread.currentActivityThread()

Warning: Because the python function is called from the PythonActivity thread, you need to be careful about your own calls.

Service (android.service)

Services of an application are controlled through the class `AndroidService`.

class `android.service.`**AndroidService**(*title*, *description*)
Run `service/main.py` from the application directory as a service.

Parameters

- **title** (*str*) – Notification title, default to 'Python service'
- **description** (*str*) – Notification text, default to 'Kivy Python service started'

start(*arg*)
Start the service.

Parameters **arg** (*str*) – Argument to pass to a service, through the environment variable `PYTHON_SERVICE_ARGUMENT`. Defaults to ''

stop()
Stop the service.

Application activity part example, `main.py`:

```
from android import AndroidService

...

 class ServiceExample(App):

  ...

     def start_service(self):
         self.service = AndroidService('Sevice example', 'service is running')
         self.service.start('Hello From Service')

     def stop_service(self):
         self.service.stop()
```

Application service part example, `service/main.py`:

```
import os
import time

# get the argument passed
arg = os.getenv('PYTHON_SERVICE_ARGUMENT')

while True:
    # this will print 'Hello From Service' continually, even when the application is switched
    print arg
    time.sleep(1)
```

1.9.4 Java API (pyjnius)

Using PyJNIus to access the Android API restricts the usage to a simple call of the **autoclass** constructor function and a second call to instantiate this class.

You can access through this method the entire Java Android API, e.g., the `DisplayMetrics` of an Android device could be fetched using the following piece of code:

```
DisplayMetrics = autoclass('android.util.DisplayMetrics')
metrics = DisplayMetrics()
metrics.setToDefaults()
self.densityDpi = metrics.densityDpi
```

You can access all fields and methods as described in the Java Android DisplayMetrics API as shown here with the method *setToDefaults()* and the field *densityDpi*. Before you use a view field, you should always call *setToDefaults* to initiate to the default values of the device.

Currently only JavaMethod, JavaStaticMethod, JavaField, JavaStaticField and JavaMultipleMethod are built into PyJNIus, therefore such constructs like registerListener or something like this must still be coded in Java. For this the Android module described below is available to access some of the hardware on Android devices.

Activity

If you want the instance of the current Activity, use:

- `PythonActivity.mActivity` if you are running an application
- `PythonService.mService` if you are running a service

class org.renpy.android.**PythonActivity**

mInfo
Instance of an ApplicationInfo

mActivity
Instance of *PythonActivity*.

registerNewIntentListener(*NewIntentListener listener*)
Register a new instance of NewIntentListener to be called when onNewIntent is called.

unregisterNewIntentListener(*NewIntentListener listener*)
Unregister a previously registered listener from *registerNewIntentListener()*

registerActivityResultListener(*ActivityResultListener listener*)
Register a new instance of ActivityResultListener to be called when onActivityResult is called.

unregisterActivityResultListener(*ActivityResultListener listener*)
Unregister a previously registered listener from *PythonActivity.registerActivityResultListener()*

class org.renpy.android.**PythonActivity_ActivityResultListener**

Note: This class is a subclass of PythonActivity, so the notation will be PythonActivity$ActivityResultListener

Listener interface for onActivityResult. You need to implementing it, create an instance and use it with *PythonActivity.registerActivityResultListener()*.

onActivityResult(*int requestCode*, *int resultCode*, *Intent data*)
Method to implement

class org.renpy.android.**PythonActivity_NewIntentListener**

Note: This class is a subclass of PythonActivity, so the notation will be PythonActivity$NewIntentListener

Listener interface for onNewIntent. You need to implementing it, create an instance and use it with registerNewIntentListener().

onNewIntent(*Intent intent*)
Method to implement

Action

class org.renpy.android.**Action**
This module is built to deliver data to someone else.

send(*mimetype*, *filename*, *subject*, *text*, *chooser_title*)
Deliver data to someone else. This method is a wrapper around ACTION_SEND

Parameters

***mimetype*: str** Must be a valid mimetype, that represent the content to sent.

***filename*: str, default to None** (optional) Name of the file to attach. Must be a absolute path.

***subject*: str, default to None** (optional) Default subject

***text*: str, default to None** (optional) Content to send.

***chooser_title*: str, default to None** (optional) Title of the android chooser window, default to 'Send email...'

Sending a simple hello world text:

```
android.action_send('text/plain', text='Hello world',
    subject='Test from python')
```

Sharing an image file:

```
# let's say you've make an image in /sdcard/image.png
android.action_send('image/png', filename='/sdcard/image.png')
```

Sharing an image with a default text too:

```
android.action_send('image/png', filename='/sdcard/image.png',
    text='Hi,\n\tThis is my awesome image, what do you think about it ?')
```

Hardware

class org.renpy.android.**Hardware**

This module is built for accessing hardware devices of an Android device. All the methods are static and public, you don't need an instance.

vibrate(*s*)

Causes the phone to vibrate for *s* seconds. This requires that your application have the VIBRATE permission.

getHardwareSensors()

Returns a string of all hardware sensors of an Android device where each line lists the informations about one sensor in the following format:

Name=name,Vendor=vendor,Version=version,MaximumRange=maximumRange,MinDelay=minDelay,Power=power,Type=

For more information about this informations look into the original Java API for the Sensors Class

accelerometerSensor

This variable links to a generic3AxisSensor instance and their functions to access the accelerometer sensor

orientationSensor

This variable links to a generic3AxisSensor instance and their functions to access the orientation sensor

magenticFieldSensor

The following two instance methods of the generic3AxisSensor class should be used to enable/disable the sensor and to read the sensor

changeStatus(*boolean enable*)

Changes the status of the sensor, the status of the sensor is enabled, if *enable* is true or disabled, if *enable* is false.

readSensor()

Returns an (x, y, z) tuple of floats that gives the sensor reading, the units depend on the sensor as shown on the Java API page for SensorEvent. The sesnor must be enabled before this function is called. If the tuple contains three zero values, the accelerometer is not enabled, not available, defective, has not returned a reading, or the device is in free-fall.

get_dpi()
Returns the screen density in dots per inch.

show_keyboard()
Shows the soft keyboard.

hide_keyboard()
Hides the soft keyboard.

wifi_scanner_enable()
Enables wifi scanning.

> **Note:** ACCESS_WIFI_STATE and CHANGE_WIFI_STATE permissions are required.

wifi_scan()
Returns a String for each visible WiFi access point

(SSID, BSSID, SignalLevel)

Further Modules

Some further modules are currently available but not yet documented. Please have a look into the code and you are very welcome to contribute to this documentation.

1.9.5 Contribute

Extending Python for android native support

So, you want to get into python-for-android and extend what's available to Python on Android ?

Turns out it's not very complicated, here is a little introduction on how to go about it. Without Pyjnius, the schema to access the Java API from Cython is:

```
[1] Cython -> [2] C JNI -> [3] Java
```

Think about acceleration sensors: you want to get the acceleration values in Python, but nothing is available natively. Lukcily you have a Java API for that : the Google API is available here http://developer.android.com/reference/android/hardware/Sensor.html

You can't use it directly, you need to do your own API to use it in python, this is done in 3 steps

Step 1: write the java code to create very simple functions to use

like : accelerometer Enable/Reading In our project, this is done in the Hardware.java: https://github.com/kivy/python-for-android/blob/master/src/src/org/renpy/android/Hardware.java you can see how it's implemented

Step 2 : write a jni wrapper

This is a C file to be able to invoke/call Java functions from C, in our case, step 2 (and 3) are done in the android python module. The JNI part is done in the android_jni.c: https://github.com/kivy/python-for-android/blob/master/recipes/android/src/android_jni.c

Step 3 : you have the java part, that you can call from the C

You can now do the Python extension around it, all the android python part is done in https://github.com/kivy/python-for-android/blob/master/recipes/android/src/android.pyx

→ [python] android.accelerometer_reading [C] android_accelerometer_reading [Java] Hardware.accelerometer_reading()

The jni part is really a C api to call java methods. a little bit hard to get it with the syntax, but working with current example should be ok

Example with bluetooth

Start directly from a fork of https://github.com/kivy/python-for-android

The first step is to identify where and how they are doing it in sl4a, it's really easy, because everything is already done as a client/server client/consumer approach, for bluetooth, they have a "Bluetooth facade" in java.

http://code.google.com/p/android-scripting/source/browse/android/BluetoothFacade/src/com/googlecode/android_scripting/facade/Blue

You can learn from it, and see how is it's can be used as is, or if you can simplify / remove stuff you don't want.

From this point, create a bluetooth file in python-for-android/tree/master/src/src/org/renpy/android in Java.

Do a good API (enough simple to be able to write the jni in a very easy manner, like, don't pass any custom java object in argument).

Then write the JNI, and then the python part.

3 steps, once you get it, the real difficult part is to write the java part :)

Jni gottchas

- package must be org.renpy.android, don't change it.

Create your own recipes

A recipe is a script that contains the "definition" of a module to compile. The directory layout of a recipe for a <modulename> is something like:

```
python-for-android/recipes/<modulename>/recipe.sh
python-for-android/recipes/<modulename>/patches/
python-for-android/recipes/<modulename>/patches/fix-path.patch
```

When building, all the recipe builds must go to:

```
python-for-android/build/<modulename>/<archiveroot>
```

For example, if you want to create a recipe for sdl, do:

```
cd python-for-android/recipes
mkdir sdl
cp recipe.sh.tmpl sdl/recipe.sh
sed -i 's#XXX#sdl#' sdl/recipe.sh
```

Then, edit the sdl/recipe.sh to adjust other information (version, url) and complete the build function.

1.9.6 Related projects

- PGS4A: http://pygame.renpy.org/ (thanks to Renpy to make it possible)
- Android scripting: http://code.google.com/p/android-scripting/
- Python on a chip: http://code.google.com/p/python-on-a-chip/

1.9.7 FAQ

arm-linux-androideabi-gcc: Internal error: Killed (program cc1)

This could happen if you are not using a validated SDK/NDK with Python for Android. Go to `prerequisites.rst` to see which one are working.

_sqlite3.so not found

We recently fixed sqlite3 compilation. In case of this error, you must:

- Install development headers for sqlite3 if they are not already installed. On Ubuntu:

  apt-get install libsqlite3-dev

- Compile the distribution with (sqlite3 must be the first argument):

```
./distribute.sh -m 'sqlite3 kivy'
```

- Go into your distribution at *dist/default*
- Edit blacklist.txt, and remove all the lines concerning sqlite3:

```
sqlite3/*
lib-dynload/_sqlite3.so
```

Then sqlite3 will be compiled and included in your APK.

Too many levels of symbolic links

Python for Android does not work within a virtual enviroment. The Python for Android directory must be outside of the virtual enviroment prior to running

./distribute.sh -m “kivy”

or else you may encounter OSError: [Errno 40] Too many levels of symbolic links.

1.10 Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

a

android, 33
android.activity, 33
android.billing, 34
android.broadcast, 35
android.mixer, 36
android.runnable, 37
android.service, 37

o

org.renpy.android, 38

Symbols

__init__() (android.broadcast.BroadcastReceiver method), 35

A

accelerometerSensor (org.renpy.android.Hardware attribute), 40
Action (class in org.renpy.android), 39
android (module), 33
android.activity (module), 33
android.billing (module), 34
android.broadcast (module), 35
android.mixer (module), 36
android.runnable (module), 37
android.service (module), 37
AndroidService (class in android.service), 37

B

bind() (in module android.activity), 33
BroadcastReceiver (class in android.broadcast), 35

C

changeStatus() (org.renpy.android.Hardware method), 40
check_pause() (in module android), 33

G

get_dpi() (org.renpy.android.Hardware method), 40
getHardwareSensors() (org.renpy.android.Hardware method), 40

H

Hardware (class in org.renpy.android), 40
hide_keyboard() (org.renpy.android.Hardware method), 41

M

magenticFieldSensor (org.renpy.android.Hardware attribute), 40
map_key() (in module android), 33

O

onActivityResult() (org.renpy.android.PythonActivity_ActivityResultListen method), 39
onNewIntent() (org.renpy.android.PythonActivity_NewIntentListener method), 39
org.renpy.android (module), 38
orientationSensor (org.renpy.android.Hardware attribute), 40

P

PythonActivity (class in org.renpy.android), 38
PythonActivity.mActivity (in module org.renpy.android), 39
PythonActivity.mInfo (in module org.renpy.android), 39
PythonActivity_ActivityResultListener (class in org.renpy.android), 39
PythonActivity_NewIntentListener (class in org.renpy.android), 39

R

readSensor() (org.renpy.android.Hardware method), 40
registerActivityResultListener() (org.renpy.android.PythonActivity method), 39
registerNewIntentListener() (org.renpy.android.PythonActivity method), 39

S

send() (org.renpy.android.Action method), 39
show_keyboard() (org.renpy.android.Hardware method), 41
start() (android.broadcast.BroadcastReceiver method), 35
start() (android.service.AndroidService method), 37
stop() (android.broadcast.BroadcastReceiver method), 35
stop() (android.service.AndroidService method), 37

U

unbind() (in module android.activity), 33
unregisterActivityResultListener() (org.renpy.android.PythonActivity method), 39

unregisterNewIntentListener()
(org.renpy.android.PythonActivity method), 39

V

vibrate() (org.renpy.android.Hardware method), 40

W

wait_for_resume() (in module android), 33
wifi_scan() (org.renpy.android.Hardware method), 41
wifi_scanner_enable() (org.renpy.android.Hardware
method), 41