python-for-android Documentation

Release 0.1

Alexander Taylor

Jun 19, 2017

Contents

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 Android 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

Getting Started

Getting up and running on python-for-android (p4a) is a simple process and should only take you a couple of minutes. We'll refer to Python for android as p4a in this documentation.

Concepts

- requirements: For p4a, your applications dependencies are requirements similar to the standard *requirements.txt*, but with one difference: p4a will search for a recipe first instead of installing requirements with pip.
- recipe: A recipe is a file that defines how to compile a requirement. Any libraries that have a Python extension *must* have a recipe in p4a, or compilation will fail. If there is no recipe for a requirement, it will be downloaded using pip.
- build: A build refers to a compiled recipe.
- distribution: A distribution is the final "build" of all your compiled requirements, as an Android project that can be turned directly into an APK. p4a can contain multiple distributions with different sets of requirements.
- bootstrap: A bootstrap is the app backend that will start your application. Your application could use SDL2 as a base, or Pygame, or a web backend like Flask with a WebView bootstrap. Different bootstraps can have different build options.

Installation

Installing p4a

p4a is now available on on Pypi, so you can install it using pip:

```
pip install python-for-android
```

You can also test the master branch from Github using:

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

Installing Dependencies

p4a has several dependencies that must be installed:

- git
- ant
- python2
- cython (can be installed via pip)
- a Java JDK (e.g. openjdk-7)
- zlib (including 32 bit)
- libncurses (including 32 bit)
- unzip
- virtualenv (can be installed via pip)
- ccache (optional)

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

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install -y build-essential git zlib1g-dev python2.7 python2.7-dev␣
↪libncurses5:i386 libstdc++6:i386 zlib1g:i386 openjdk-7-jdk unzip ant ccache
```

On Arch Linux (64 bit) you should be able to run the following to install most of the dependencies (note: this list may not be complete). gcc-multilib will conflict with (and replace) gcc if not already installed. If your installation is already 32-bit, install the same packages but without `lib32-` or `-multilib`:

```
sudo pacman -S jdk7-openjdk python2 python2-pip python2-kivy mesa-libgl lib32-mesa-
↪libgl lib32-sdl2 lib32-sdl2_image lib32-sdl2_mixer sdl2_ttf unzip gcc-multilib gcc-
↪libs-multilib
```

Installing Android SDK

You need to download and unpack the Android SDK and NDK to a directory (let's say $HOME/Documents/):

- Android SDK
- Android NDK

Then, you can edit your `~/.bashrc` or other favorite shell to include new environment variables necessary for building on android:

```
# Adjust the paths!
export ANDROIDSDK="$HOME/Documents/android-sdk-21"
export ANDROIDNDK="$HOME/Documents/android-ndk-r10e"
export ANDROIDAPI="14"  # Minimum API version your application require
export ANDROIDNDKVER="r10e"  # Version of the NDK you installed
```

You have the possibility to configure on any command the PATH to the SDK, NDK and Android API using:

- `--sdk_dir PATH` as an equivalent of *$ANDROIDSDK*
- `--ndk_dir PATH` as an equivalent of *$ANDROIDNDK*
- `--android_api VERSION` as an equivalent of *$ANDROIDAPI*
- `--ndk_ver PATH` as an equivalent of *$ANDROIDNDKVER*

Usage

Build a Kivy application

To build your application, you need to have a name, version, a package identifier, and explicitly write the bootstrap you want to use, as well as the requirements:

```
p4a apk --private $HOME/code/myapp --package=org.example.myapp --name "My application
↪" --version 0.1 --bootstrap=sdl2 --requirements=python2,kivy
```

This will first build a distribution that contains *python2* and *kivy*, and using a SDL2 bootstrap. Python2 is here explicitely written as kivy can work with python2 or python3.

You can also use `--bootstrap=pygame`, but this bootstrap is deprecated for use with Kivy and SDL2 is preferred.

Build a WebView application

To build your application, you need to have a name, version, a package identifier, and explicitly use the webview bootstrap, as well as the requirements:

```
p4a apk --private $HOME/code/myapp --package=org.example.myapp --name "My WebView␣
↪Application" --version 0.1 --bootstrap=webview --requirements=flask --port=5000
```

You can also replace flask with another web framework.

Replace `--port=5000` with the port on which your app will serve a website. The default for Flask is 5000.

Build an SDL2 based application

This includes e.g. PySDL2.

To build your application, you need to have a name, version, a package identifier, and explicitly write the sdl2 bootstrap, as well as the requirements:

```
p4a apk --private $HOME/code/myapp --package=org.example.myapp --name "My SDL2␣
↪application" --version 0.1 --bootstrap=sdl2 --requirements=your_requirements
```

Add your required modules in place of `your_requirements`, e.g. `--requirements=pysdl2` or `--requirements=vispy`.

Other options

You can pass other command line arguments to control app behaviours such as orientation, wakelock and app permissions. See *Bootstrap options*.

Rebuild everything

If anything goes wrong and you want to clean the downloads and builds to retry everything, run:

```
p4a clean_all
```

If you just want to clean the builds to avoid redownloading dependencies, run:

```
p4a clean_builds && p4a clean_dists
```

Getting help

If something goes wrong and you don't know how to fix it, add the `--debug` option and post the output log to the kivy-users Google group or irc channel #kivy at irc.freenode.net .

See *Troubleshooting* for more information.

Advanced usage

Recipe management

You can see the list of the available recipes with:

```
p4a recipes
```

If you are contributing to p4a and want to test a recipes again, you need to clean the build and rebuild your distribution:

```
p4a clean_recipe_build RECIPENAME
p4a clean_dists
# then rebuild your distribution
```

You can write "private" recipes for your application, just create a `p4a-recipes` folder in your build directory, and place a recipe in it (edit the `__init__.py`):

```
mkdir -p p4a-recipes/myrecipe
touch p4a-recipes/myrecipe/__init__.py
```

Distribution management

Every time you start a new project, python-for-android will internally create a new distribution (an Android build project including Python and your other dependencies compiled for Android), according to the requirements you added on the command line. You can force the reuse of an existing distribution by adding:

```
p4a apk --dist_name=myproject ...
```

This will ensure your distribution will always be built in the same directory, and avoids using more disk space every time you adjust a requirement.

You can list the available distributions:

```
p4a distributions
```

And clean all of them:

```
p4a clean_dists
```

Configuration file

python-for-android checks in the current directory for a configuration file named `.p4a`. If found, it adds all the lines as options to the command line. For example, you can add the options you would always include such as:

```
--dist_name my_example
--android_api 19
--requirements kivy,openssl
```

Going further

See the other pages of this doc for more information on specific topics:

- *Build options*
- *Commands*
- *Recipes*
- *Bootstraps*
- *Working on Android*
- *Troubleshooting*
- *Launcher*
- *Contributing*

Build options

This page contains instructions for using different build options.

Python versions

python2

Select this by adding it in your requirements, e.g. `--requirements=python2`.

This option builds Python 2.7.2 for your selected Android architecture. There are no special requirements, all the building is done locally.

The python2 build is also the way python-for-android originally worked, even in the old toolchain.

python3

Warning: Python3 support is experimental, and some of these details may change as it is improved and fully stabilised.

Note: You must manually download the CrystaX NDK and tell python-for-android to use it with `--ndk-dir /path/to/NDK`.

Select this by adding the `python3crystax` recipe to your requirements, e.g. `--requirements=python3crystax`.

This uses the prebuilt Python from the CrystaX NDK, a drop-in replacement for Google's official NDK which includes many improvements. You *must* use the CrystaX NDK 10.3.0 or higher when building with python3. You can get it here.

The python3crystax build is handled quite differently to python2 so there may be bugs or surprising behaviours. If you come across any, feel free to open an issue.

Bootstrap options

python-for-android supports multiple app backends with different types of interface. These are called *bootstraps*.

Currently the following bootstraps are supported, but we hope that it should be easy to add others if your project has different requirements. Let us know if you'd like help adding a new one.

sdl2

Use this with `--bootstrap=sdl2`, or just include the `sdl2` recipe, e.g. `--requirements=sdl2,python2`.

SDL2 is a popular cross-platform depelopment library, particularly for games. It has its own Android project support, which python-for-android uses as a bootstrap, and to which it adds the Python build and JNI code to start it.

From the point of view of a Python program, SDL2 should behave as normal. For instance, you can build apps with Kivy or PySDL2 and have them work with this bootstrap. It should also be possible to use e.g. pygame_sdl2, but this would need a build recipe and doesn't yet have one.

Build options

The sdl2 bootstrap supports the following additional command line options (this list may not be exhaustive):

- `--private`: The directory containing your project files.
- `--package`: The Java package name for your project. Choose e.g. `org.example.yourapp`.
- `--name`: The app name.
- `--version`: The version number.
- `--orientation`: Usually one of `portait`, `landscape`, `sensor` to automatically rotate according to the device orientation, or `user` to do the same but obeying the user's settings. The full list of valid options is given under `android:screenOrientation` in the Android documentation.
- `--icon`: A path to the png file to use as the application icon.
- `-- permission`: A permission name for the app, e.g. `--permission VIBRATE`. For multiple permissions, add multiple `--permission` arguments.
- `--meta-data`: Custom key=value pairs to add in the application metadata.
- `--presplash`: A path to the image file to use as a screen while the application is loading.

- `--presplash-color`: The presplash screen background color, of the form `#RRGGBB` or a color name `red`, `green`, `blue` etc.
- `--wakelock`: If the argument is included, the application will prevent the device from sleeping.
- `--window`: If the argument is included, the application will not cover the Android status bar.
- `--blacklist`: The path to a file containing blacklisted patterns that will be excluded from the final APK. Defaults to `./blacklist.txt`.
- `--whitelist`: The path to a file containing whitelisted patterns that will be included in the APK even if also blacklisted.
- `--add-jar`: The path to a .jar file to include in the APK. To include multiple jar files, pass this argument multiple times.
- `--intent-filters`: A file path containing intent filter xml to be included in AndroidManifest.xml.
- `--service`: A service name and the Python script it should run. See *Arbitrary service scripts*.
- `--add-source`: Add a source directory to the app's Java code.
- `--no-compile-pyo`: Do not optimise .py files to .pyo.

webview

You can use this with `--bootstrap=webview`, or include the `webviewjni` recipe, e.g. `--requirements=webviewjni,python2`.

The webview bootstrap gui is, per the name, a WebView displaying a webpage, but this page is hosted on the device via a Python webserver. For instance, your Python code can start a Flask application, and your app will display and allow the user to navigate this website.

Note: Your Flask script must start the webserver *without* :code:`debug=True`. Debug mode doesn't seem to work on Android due to use of a subprocess.

This bootstrap will automatically try to load a website on port 5000 (the default for Flask), or you can specify a different option with the *–port* command line option. If the webserver is not immediately present (e.g. during the short Python loading time when first started), it will instead display a loading screen until the server is ready.

- `--private`: The directory containing your project files.
- `--package`: The Java package name for your project. Choose e.g. `org.example.yourapp`.
- `--name`: The app name.
- `--version`: The version number.
- `--orientation`: Usually one of `portait`, `landscape`, `sensor` to automatically rotate according to the device orientation, or `user` to do the same but obeying the user's settings. The full list of valid options is given under `android:screenOrientation` in the Android documentation.
- `--icon`: A path to the png file to use as the application icon.
- `-- permission`: A permission name for the app, e.g. `--permission VIBRATE`. For multiple permissions, add multiple `--permission` arguments.
- `--meta-data`: Custom key=value pairs to add in the application metadata.
- `--presplash`: A path to the image file to use as a screen while the application is loading.

- `--presplash-color`: The presplash screen background color, of the form `#RRGGBB` or a color name `red`, `green`, `blue` etc.
- `--wakelock`: If the argument is included, the application will prevent the device from sleeping.
- `--window`: If the argument is included, the application will not cover the Android status bar.
- `--blacklist`: The path to a file containing blacklisted patterns that will be excluded from the final APK. Defaults to `./blacklist.txt`.
- `--whitelist`: The path to a file containing whitelisted patterns that will be included in the APK even if also blacklisted.
- `--add-jar`: The path to a .jar file to include in the APK. To include multiple jar files, pass this argument multiple times.
- `--intent-filters`: A file path containing intent filter xml to be included in AndroidManifest.xml.
- `--service`: A service name and the Python script it should run. See *Arbitrary service scripts*.
- `add-source`: Add a source directory to the app's Java code.
- `--port`: The port on localhost that the WebView will access. Defaults to 5000.

pygame

You can use this with `--bootstrap=pygame`, or simply include the `pygame` recipe in your `--requirements`.

The pygame bootstrap is the original backend used by Kivy, and still works fine for use with Kivy apps. It may also work for pure pygame apps, but hasn't been developed with this in mind.

This bootstrap will eventually be deprecated in favour of sdl2, but not before the sdl2 bootstrap includes all the features that would be lost.

Build options

The pygame bootstrap supports the following additional command line options (this list may not be exhaustive):

- `--private`: The directory containing your project files.
- `--dir`: The directory containing your project files if you want them to be unpacked to the external storage directory rather than the app private directory.
- `--package`: The Java package name for your project. Choose e.g. `org.example.yourapp`.
- `--name`: The app name.
- `--version`: The version number.
- `--orientation`: One of `portait`, `landscape` or `sensor` to automatically rotate according to the device orientation.
- `--icon`: A path to the png file to use as the application icon.
- `--ignore-path`: A path to ignore when including the app files. Pass multiple times to ignore multiple paths.
- `-- permission`: A permission name for the app, e.g. `--permission VIBRATE`. For multiple permissions, add multiple `--permission` arguments.
- `--meta-data`: Custom key=value pairs to add in the application metadata.
- `--presplash`: A path to the image file to use as a screen while the application is loading.
- `--wakelock`: If the argument is included, the application will prevent the device from sleeping.

- `--window`: If the argument is included, the application will not cover the Android status bar.
- `--blacklist`: The path to a file containing blacklisted patterns that will be excluded from the final APK. Defaults to `./blacklist.txt`.
- `--whitelist`: The path to a file containing whitelisted patterns that will be included in the APK even if also blacklisted.
- `--add-jar`: The path to a .jar file to include in the APK. To include multiple jar files, pass this argument multiple times.
- `--intent-filters`: A file path containing intent filter xml to be included in AndroidManifest.xml.
- `--service`: A service name and the Python script it should run. See *Arbitrary service scripts*.
- `add-source`: Add a source directory to the app's Java code.
- `--compile-pyo`: Optimise .py files to .pyo.
- `--resource`: A key=value pair to add in the string.xml resource file.

Commands

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

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).

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.

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.

--arch The architecture to build for. Currently only one architecture can be targeted at a time, and a given distribution can only include one architecture.

--bootstrap BOOTSTRAP The Java bootstrap to use for your application. You mostly don't need to worry about this or set it manually, as an appropriate bootstrap will be chosen from your `--requirements`. Current choices are `sdl2` or `pygame`; `sdl2` is experimental but preferable where possible.

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

---

distutils/setuptools integration

Instead of running p4a via the command line, you can integrate with distutils and setup.py.

The base command is:

```
python setup.py apk
```

The files included in the APK will be all those specified in the `package_data` argument to setup. For instance, the following example will include all .py and .png files in the `testapp` folder:

```
from distutils.core import setup
from setup

setup(
    name='testapp_setup',
    version='1.1',
    description='p4a setup.py example',
    author='Your Name',
    author_email='youremail@address.com',
    packages=find_packages(),
    options=options,
    package_data={'testapp': ['*.py', '*.png']}
)
```

The app name and version will also be read automatically from the setup.py.

The Android package name uses `org.test.lowercaseappname` if not set explicitly.

The `--private` argument is set automatically using the package_data, you should *not* set this manually.

The target architecture defaults to `--armeabi`.

All of these automatic arguments can be overridden by passing them manually on the command line, e.g.:

```
python setup.py apk --name="Testapp Setup" --version=2.5
```

Adding p4a arguments in setup.py

Instead of providing extra arguments on the command line, you can store them in setup.py by passing the `options` parameter to `setup`. For instance:

```
from distutils.core import setup
from setuptools import find_packages

options = {'apk': {'debug': None,  # use None for arguments that don't pass a value
                   'requirements': 'sdl2,pyjnius,kivy,python2',
                   'android-api': 19,
                   'ndk-dir': '/path/to/ndk',
                   'dist-name': 'bdisttest',
                   }}

packages = find_packages()
print('packages are', packages)

setup(
    name='testapp_setup',
    version='1.1',
    description='p4a setup.py example',
    author='Your Name',
    author_email='youremail@address.com',
    packages=find_packages(),
    options=options,
    package_data={'testapp': ['*.py', '*.png']}
)
```

These options will be automatically included when you run `python setup.py apk`. Any options passed on the command line will override these values.

Adding p4a arguments in setup.cfg

You can also provide p4a arguments in the setup.cfg file, as normal for distutils. The syntax is:

```
[apk]

argument=value

requirements=sdl2,kivy
```

Recipes

This page describes how python-for-android (p4a) compilation recipes work, and how to build your own. If you just want to build an APK, ignore this and jump straight to the *Getting Started*.

Recipes are special scripts for compiling and 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. No recipe is necessary to use of Python modules with no compiled components; these are installed automaticaly via pip.

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.

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'

    patches = ['some_fix.patch']  # Paths relative to the recipe dir

    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.

You must include `recipe = YourRecipe()`. This variable is accessed when the recipe is imported.

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
```

These methods are always run in the listed order; prebuild, then build, then postbuild.

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.

Methods and tools to help with compilation

Patching modules before installation

You can easily apply patches to your recipes by adding them to the `patches` declaration, e.g.:

```
patches = ['some_fix.patch',
           'another_fix.patch']
```

The paths should be relative to the recipe file. Patches are automatically applied just once (i.e. not reapplied the second time python-for-android is run).

You can also use the helper functions in `pythonforandroid.patching` to apply patches depending on certain conditions, e.g.:

```
from pythonforandroid.patching import will_build, is_arch

...

class YourRecipe(Recipe):

    patches = [('x86_patch.patch', is_arch('x86')),
               ('sdl2_compatibility.patch', will_build('sdl2'))]

    ...
```

You can include your own conditions by passing any function as the second entry of the tuple. It will receive the `arch` (e.g. x86, armeabi) and `recipe` (i.e. the Recipe object) as kwargs. The patch will be applied only if the function returns True.

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 from 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 for each architecture 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, arch):
    name = self.site_packages_name
    if name is None:
        name = self.name
    if self.ctx.has_package(name):
        info('Python package already exists in site-packages')
        return False
    info('{} apparently isn\'t already in site-packages'.format(name))
    return True
```

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.

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.

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).

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.

A Recipe template

The following template includes all the recipe sections you might use. 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'
    # {version} will be replaced with self.version when downloading

    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()
```

Examples of recipes

This documentation covers most of what is ever necessary to make a recipe work. For further examples, python-for-android includes many recipes for popular modules, which are an excellent resource to find out how to add your own. You can find these in the python-for-android Github page.

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.

Bootstraps

This page is about creating new bootstrap backends. For build options of existing bootstraps (i.e. with SDL2, Pygame, Webview etc.), see *build options*.

python-for-android (p4a) supports multiple *bootstraps*. These fulfill 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.

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.

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, ArchAndroid, logger, info_main, which
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 create 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.

Services

python-for-android supports the use of Android Services, background tasks running in separate processes. These are the closest Android equivalent to multiprocessing on e.g. desktop platforms, and it is not possible to use normal multiprocessing on Android. Services are also the only way to run code when your app is not currently opened by the user.

Services must be declared when building your APK. Each one will have its own main.py file with the Python script to be run. You can communicate with the service process from your app using e.g. osc or (a heavier option) twisted.

Service creation

There are two ways to have services included in your APK.

Service folder

This basic method works with both the new SDL2 and old Pygame bootstraps. It is recommended to use the second method (below) where possible.

Create a folder named `service` in your app directory, and add a file `service/main.py`. This file should contain the Python code that you want the service to run.

To start the service, use the `start_service` function from the `android` module (included automatically with the Pygame bootstrap, you must add it to the requirements manually with SDL2 if you wish to use this method):

```
import android
android.start_service(title='service name',
                      description='service description',
                      arg='argument to service')
```

Arbitrary service scripts

Note: This service method is *not supported* by the Pygame bootstrap.

This method is recommended for non-trivial use of services as it is more flexible, supporting multiple services and a wider range of options.

To create the service, create a python script with your service code and add a `--service=myservice:/path/to/myservice.py` argument when calling python-for-android. The `myservice` name before the colon is the name of the service class, via which you will interact with it later. You can add multiple `--service` arguments to include multiple services, which you will later be able to stop and start from your app.

To run the services (i.e. starting them from within your main app code), you must use PyJNIus to interact with the java class python-for-android creates for each one, as follows:

```
from jnius import autoclass
service = autoclass('your.package.name.ServiceMyservice')
mActivity = autoclass('org.kivy.android.PythonActivity').mActivity
argument = ''
service.start(mActivity, argument)
```

Here, `your.package.name` refers to the package identifier of your APK as set by the `--package` argument to python-for-android, and the name of the service is `ServiceYourservicename`, in which `Yourservicename` is the identifier passed to the `--service` argument with the first letter upper case. You must also pass the `argument` parameter even if (as here) it is an empty string. If you do pass it, the service can make use of this argument.

Services support a range of options and interactions not yet documented here but all accessible via calling other methods of the `service` reference.

Note: The app root directory for Python imports will be in the app

root folder even if the service file is in a subfolder. To import from your service folder you must use e.g. `import service.module` instead of `import module`, if the service file is in the `service/` folder.

Working on Android

This page gives details on accessing Android APIs and managing other interactions on Android.

Accessing Android APIs

When writing an Android application you may want to access the normal Android Java APIs, in order to control your application's appearance (fullscreen, orientation etc.), interact with other apps or use hardware like vibration and sensors.

You can access these with Pyjnius, a Python library for automatically wrapping Java and making it callable from Python code. Pyjnius is fairly simple to use, but not very Pythonic and it 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; you can call the same code in Python but have it do the right thing also on platforms other than Android.

Pyjnius and Plyer are independent projects whose documentation is linked above. See below for some simple introductory examples, and explanation of how to include these modules in your APKs.

This page also documents the `android` module which you can include with p4a, but this is mostly replaced by Pyjnius and is not recommended for use in new applications.

Using Pyjnius

Pyjnius lets you call the Android API directly from Python 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 *pyjnius* to your build requirements, e.g. `--requirements=flask,pyjnius`. 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.

Using Plyer

Plyer provides a much less verbose, Pythonic wrapper to platform-specific APIs. It supports Android as well as iOS and desktop operating systems, though plyer is a work in progress and not all platforms support all Plyer calls yet.

Plyer 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, e.g. `--requirements=plyer`.

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 than with Pyjnius!

Using `android`

This Cython module was used for Android API interaction with Kivy's old interface, but is now mostly replaced by Pyjnius.

The `android` Python module can be included by adding it to your requirements, e.g. `--requirements=kivy, android`. It is not automatically included by Kivy unless you use the old (Pygame) bootstrap.

This module is not separately documented. You can read the source on Github.

One useful facility of this module is to make `webbrowser.open()` work on Android. You can replicate this effect without using the android module via the following code:

```
from jnius import autoclass

def open_url(url):
Intent = autoclass('android.content.Intent')
Uri = autoclass('android.net.Uri')
browserIntent = Intent()
browserIntent.setAction(Intent.ACTION_VIEW)
browserIntent.setData(Uri.parse(url))
currentActivity = cast('android.app.Activity', mActivity)
currentActivity.startActivity(browserIntent)

class AndroidBrowser(object):
    def open(self, url, new=0, autoraise=True):
        open_url(url)
    def open_new(self, url):
        open_url(url)
    def open_new_tab(self, url):
```

```
        open_url(url)

import webbrowser
webbrowser.register('android', AndroidBrowser, None, -1)
```

Working with the App lifecycle

Dismissing the splash screen

With the SDL2 bootstrap, the app's splash screen may not be dismissed immediately when your app has finished loading, due to a limitation with the way we check if the app has properly started. In this case, the splash screen overlaps the app gui for a short time.

You can dismiss the splash screen as follows. Run this code from your app build method (or use `kivy.clock.Clock.schedule_once` to run it in the following frame):

```
from jnius import autoclass
activity = autoclass('org.kivy.android.PythonActivity').mActivity
activity.removeLoadingScreen()
```

This problem does not affect the Pygame bootstrap, as it uses a different splash screen method.

Handling the back button

Android phones always have a back button, which users expect to perform an appropriate in-app function. If you do not handle it, Kivy apps will actually shut down and appear to have crashed.

In SDL2 bootstraps, the back button appears as the escape key (keycode 27, codepoint 270). You can handle this key to perform actions when it is pressed.

For instance, in your App class in Kivy:

```
from kivy.core.window import Window

class YourApp(App):

   def build(self):
      Window.bind(on_keyboard=self.key_input)
      return Widget() # your root widget here as normal

   def key_input(self, window, key, scancode, codepoint, modifier):
      if key == 27:
         return True  # override the default behaviour
      else:           # the key now does nothing
         return False
```

Pausing the App

When the user leaves an App, it is automatically paused by Android, although it gets a few seconds to store data etc. if necessary. Once paused, there is no guarantee that your app will run again.

With Kivy, add an `on_pause` method to your App class, which returns True:

```
def on_pause(self):
    return True
```

With the webview bootstrap, pausing should work automatically.

Under SDL2, you can handle the appropriate events (see SDL_APP_WILLENTERBACKGROUND etc.).

Troubleshooting

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.

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.

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.

Unpacking an APK

It is sometimes useful to unpack a pacakged APK to see what is inside, especially when debugging python-for-android itself.

APKs are just zip files, so you can extract the contents easily:

```
unzip YourApk.apk
```

At the top level, this will always contain the same set of files:

```
$ ls
AndroidManifest.xml  classes.dex  META-INF     res
assets               lib          YourApk.apk  resources.arsc
```

The Python distribution is in the assets folder:

```
$ cd assets
$ ls
private.mp3
```

`private.mp3` is actually a tarball containing all your packaged data, and the Python distribution. Extract it:

```
$ tar xf private.mp3
```

This will reveal all the Python-related files:

```
$ ls
android_runnable.pyo  include          interpreter_subprocess  main.kv   pipinterface.
↪kv   settings.pyo
assets                __init__.pyo     interpreterwrapper.pyo  main.pyo  pipinterface.
↪pyo  utils.pyo
editor.kv             interpreter.kv   lib                     menu.kv   private.mp3 ␣
↪     widgets.pyo
editor.pyo            interpreter.pyo  libpymodules.so         menu.pyo  settings.kv
```

Most of these files have been included by the user (in this case, they come from one of my own apps), the rest relate to the python distribution.

With Python 2, the Python installation can mostly be found in the `lib` folder. With Python 3 (using the `python3crystax` recipe), the Python installation can be found in a folder named `crystax_python`.

Common errors

The following are common problems and resolutions that users have reported.

AttributeError: 'AnsiCodes' object has no attribute 'LIGHTBLUE_EX'

This occurs if your version of colorama is too low, install version 0.3.3 or higher.

If you install python-for-android with pip or via setup.py, this dependency should be taken care of automatically.

AttributeError: 'Context' object has no attribute 'hostpython'

This is a known bug in some releases. To work around it, add your python requirement explicitly, e.g. `--requirements=python2,kivy`. This also applies when using buildozer, in which case add python2 to your buildozer.spec requirements.

linkname too long

This can happen when you try to include a very long filename, which doesn't normally happen but can occur accidentally if the p4a directory contains a .buildozer directory that is not excluded from the build (e.g. if buildozer was previously used). Removing this directory should fix the problem, and is desirable anyway since you don't want it in the APK.

Exception in thread "main" java.lang.UnsupportedClassVersionError: com/android/dx/command/Main : Unsupported major.minor version 52.0

This occurs due to a java version mismatch, it should be fixed by installing Java 8 (e.g. the openjdk-8-jdk package on Ubuntu).

JNI DETECTED ERROR IN APPLICATION: static jfieldID 0x0000000 not valid for class java.lang.Class<org.renpy.android.PythonActivity>

This error appears in the logcat log if you try to access `org.renpy.android.PythonActivity` from within the new toolchain. To fix it, change your code to reference `org.kivy.android.PythonActivity` instead.

Launcher

The Kivy Launcher is an Android application that can run any Kivy app stored in *kivy* folder on SD Card. You can download the latest stable version for your android device from the Play Store.

The stable launcher comes with various Python packages and permissions, usually listed in the description in the store. Those aren't always enough for an application to run or even launch if you work with other dependencies that are not packaged.

The Kivy Launcher is intended for quick and simple testing, for anything more advanced we recommend building your own APK with python-for-android.

Building

The Kivy Launcher is built using python-for-android. To get the most recent versions of packages you need to clean them first, so that the packager won't grab an old (cached) package instead of fresh one.

```
p4a clean_download_cache requirements
p4a clean_dists && p4a clean_builds
p4a apk --requirements=requirements \
        --permission PERMISSION \
        --package=the.package.name \
        --name="App name" \
        --version=x.y.z \
        --android_api XY \
        --bootstrap=pygame or sdl2 \
```

```
        --launcher \
        --minsdk 13
```

Note: *–minsdk 13* is necessary for the new toolchain, otherwise you'll be able to run apps only in *landscape* orientation.

Warning: Do not use any of *–private*, *–public*, *–dir* or other arguments for adding *main.py* or *main.pyo* to the app. The argument *–launcher* is above them and tells the p4a to build the launcher version of the APK.

Usage

Once the launcher is installed, you need to create a folder in your external storage directory (e.g. `/storage/emulated/0` or `/sdcard`) - this is normally your 'home' directory in a file browser. Each new folder inside *kivy* represents a separate application:

```
/sdcard/kivy/<yourapplication>
```

Each application folder must contain an *android.txt* file. The file has to contain three basic lines:

```
title=<Application Title>
author=<Your Name>
orientation=<portrait|landscape>
```

The file is editable so you can change for example orientation or name. These are the only options dynamically configurable here, although when the app runs you can call the Android API with PyJNIus to change other settings.

After you set your *android.txt* file, you can now run the launcher and start any available app from the list.

To differentiate between apps in `/sdcard/kivy` you can include an icon named `icon.png` to the folder. The icon should be a square.

Release on the market

Launcher is released on Google Play with each new Kivy stable branch. The master branch is not suitable for a regular user because it changes quickly and needs testing.

Source code

If you feel confident, feel free to improve the launcher. You can find the source code at pygame org.renpy.android or at **|kivy|_**.

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.

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

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 !
```

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:

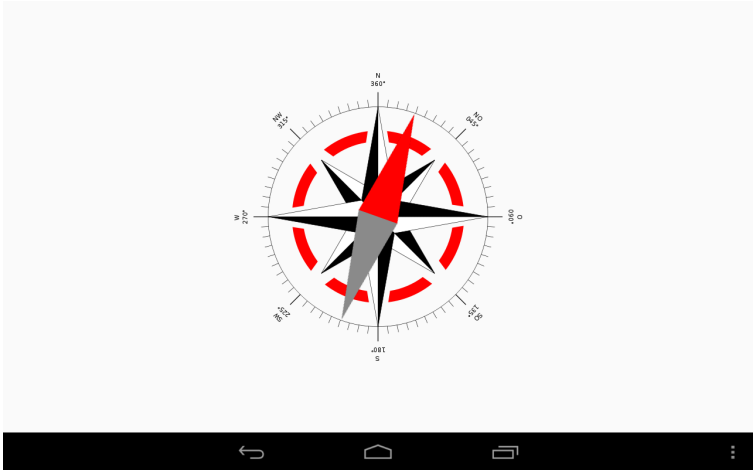

Fig. 1.1: Screenshot of the Kivy Compass App (Source of the Compass Windrose: Wikipedia)

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:

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.

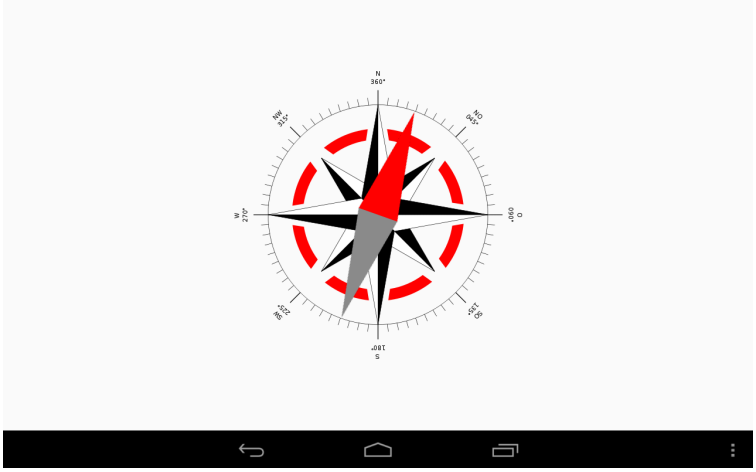

Fig. 1.2: Screenshot of the Kivy Compass App (Source of the Compass Windrose: Wikipedia)

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)
```

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.

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/BluetoothFacade.java

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.

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/

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* 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.

Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

o

Index

Symbols

A

B

C

G

H

M

O

P

R

S

U

V

W