python-for-android Documentation

Release 0.1

Alexander Taylor

Jul 22, 2022

Contents

python-for-android is an open source build tool to let you package Python code into standalone android APKs. These 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 apps for Android.

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

CHAPTER 1

Contents

1.1 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.

1.1.1 Concepts

Basic:

- **requirements:** For p4a, all your app's dependencies must be specified via `--requirements` similar to the standard *requirements.txt*. (Unless you specify them via a *setup.py/install_requires*) All dependencies will be mapped to "recipes" if any exist, so that many common libraries will just work. See "recipe" below for details.
- **distribution:** A distribution is the final "build" of your compiled project + requirements, as an Android project assembled by p4a that can be turned directly into an APK. p4a can contain multiple distributions with different sets of requirements.
- **build:** A build refers to a compiled recipe or distribution.
- **bootstrap:** A bootstrap is the app backend that will start your application. The default for graphical applications is SDL2. You can also use e.g. the webview for web apps, or service_only/service_library for background services. Different bootstraps have different additional build options.

Advanced:

- **recipe:** A recipe is a file telling p4a how to install a requirement that isn't by default fully Android compatible. This is often necessary for Cython or C/C++-using python extensions. p4a has recipes for many common libraries already included, and any dependency you specified will be automatically mapped to its recipe. If a dependency doesn't work and has no recipe included in p4a, then it may need one to work.

1.1.2 Installation

Installing p4a

p4a is now available 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:

- ant
- autoconf (for libffi and other recipes)
- automake
- ccache (optional)
- cmake (required for some native code recipes like jpeg's recipe)
- cython (can be installed via pip)
- gcc
- git
- libncurses (including 32 bit)
- libtool (for libffi and recipes)
- libssl-dev (for TLS/SSL support on hostpython3 and recipe)
- openjdk-8
- patch
- python3
- unzip
- virtualenv (can be installed via pip)
- zlib (including 32 bit)
- zip

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 ccache git zlib1g-dev python3 python3-dev␣
↪libncurses5:i386 libstdc++6:i386 zlib1g:i386 openjdk-8-jdk unzip ant ccache␣
↪autoconf libtool libssl-dev
```

On Arch Linux you should be able to run the following to install most of the dependencies (note: this list may not be complete):

```
sudo pacman -S core/autoconf core/automake core/gcc core/make core/patch core/pkgconf␣
↪extra/cmake extra/jdk8-openjdk extra/python-pip extra/unzip extra/zip
```

On macOS:

```
brew install autoconf automake libtool openssl pkg-config
brew tap homebrew/cask-versions
brew install --cask homebrew/cask-versions/adoptopenjdk8
```

Installing Android SDK

Warning: python-for-android is often picky about the **SDK/NDK versions.** Pick the recommended ones from below to avoid problems.

Basic SDK install

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

- Android SDK
- Android NDK

For the Android SDK, you can download 'just the command line tools'. When you have extracted these you'll see only a directory named `tools`, and you will need to run extra commands to install the SDK packages needed.

For Android NDK, note that modern releases will only work on a 64-bit operating system. **The minimal, and recommended, NDK version to use is r23b:**

- Go to ndk downloads page
- Windows users should create a virtual machine with an GNU Linux os installed, and then you can follow the described instructions from within your virtual machine.

Platform and build tools

First, install an API platform to target. **The recommended *target* API level is 27**, you can replace it with a different number but keep in mind other API versions are less well-tested and older devices are still supported down to the **recommended specified *minimum* API/NDK API level 21**:

```
$SDK_DIR/tools/bin/sdkmanager "platforms;android-27"
```

Second, install the build-tools. You can use `$SDK_DIR/tools/bin/sdkmanager --list` to see all the possibilities, but 28.0.2 is the latest version at the time of writing:

```
$SDK_DIR/tools/bin/sdkmanager "build-tools;28.0.2"
```

Configure p4a to use your SDK/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="$HOME/Documents/android-sdk-27"
export ANDROIDNDK="$HOME/Documents/android-ndk-r23b"
export ANDROIDAPI="27"  # Target API version of your application
```

(continues on next page)

(continued from previous page)

```
export NDKAPI="21"  # Minimum supported API version of your application
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 *$ANDROIDAPI*
- `--ndk-api VERSION` as an equivalent of *$NDKAPI*
- `--ndk-version VERSION` as an equivalent of *$ANDROIDNDKVER*

1.1.3 Usage

Build a Kivy or SDL2 application

To build your application, you need to specify name, version, a package identifier, the bootstrap you want to use (*sdl2* for kivy or sdl2 apps) and the requirements:

```
p4a apk --private $HOME/code/myapp --package=org.example.myapp --name "My application
↪" --version 0.1 --bootstrap=sdl2 --requirements=python3,kivy
```

Note on `--requirements`: you must add all libraries/dependencies your app needs to run. Example: `--requirements=python3,kivy,vispy`. For an SDL2 app, *kivy* is not needed, but you need to add any wrappers you might use (e.g. *pysdl2*).

This *p4a apk ...* command builds a distribution with *python3*, *kivy*, and everything else you specified in the requirements. It will be packaged using a SDL2 bootstrap, and produce an *.apk* file.

Compatibility notes:

- Python 2 is no longer supported by python-for-android. The last release supporting Python 2 was v2019.10.06.

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
```

Please note as with kivy/SDL2, you need to specify all your additional requirements/dependencies.

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 a Service library archive

To build an android archive (.aar), containing an android service , you need a name, version, package identifier, explicitly use the service_library bootstrap, and declare service entry point (See *services* for more options), as well as the requirements and arch(s):

```
p4a aar --private $HOME/code/myapp --package=org.example.myapp --name "My library" --
↪version 0.1 --bootstrap=service_library --requirements=python3 --release --
↪service=myservice:service.py --arch=arm64-v8a --arch=armeabi-v7a
```

You can then call the generated Java entrypoint(s) for your Python service(s) in other apk build frameworks.

Exporting the Android App Bundle (aab) for distributing it on Google Play

Starting from August 2021 for new apps and from November 2021 for updates to existings apps, Google Play Console will require the Android App Bundle instead of the long lived apk.

python-for-android handles by itself the needed work to accomplish the new requirements:

> p4a aab –private $HOME/code/myapp –package=org.example.myapp –name=”My App” –version 0.1 –bootstrap=sdl2 –requirements=python3,kivy –arch=arm64-v8a –arch=armeabi-v7a –release

This *p4a aab ...* command builds a distribution with *python3*, *kivy*, and everything else you specified in the requirements. It will be packaged using a SDL2 bootstrap, and produce an *.aab* file that contains binaries for both *armeabi-v7a* and *arm64-v8a* ABIs.

The Android App Bundle, is supposed to be used for distributing your app. If you need to test it locally, on your device, you can use *bundletool <https://developer.android.com/studio/command-line/bundletool>*

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 the kivy #support Discord channel.

See *Troubleshooting* for more information.

1.1.4 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 27
--requirements kivy,openssl
```

Overriding recipes sources

You can override the source of any recipe using the `$P4A_recipename_DIR` environment variable. For instance, to test your own Kivy branch you might set:

```
export P4A_kivy_DIR=/home/username/kivy
```

The specified directory will be copied into python-for-android instead of downloading from the normal url specified in the recipe.

setup.py file (experimental)

If your application is also packaged for desktop using *setup.py*, you may want to use your *setup.py* instead of the `--requirements` option to avoid specifying things twice. For that purpose, check out *distutils/setuptools integration*

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*
- *Development and Contributing*

1.2 Build options

This page contains instructions for using different build options.

1.2.1 Python versions

python-for-android supports using Python 3.7 or higher. To explicitly select a Python version in your requirements, use e.g. `--requirements=python3==3.7.1,hostpython3==3.7.1`.

The last python-for-android version supporting Python2 was v2019.10.06

Python-for-android no longer supports building for Python 3 using the CrystaX NDK. The last python-for-android version supporting CrystaX was 0.7.0

1.2.2 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,python3`.

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. 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.
- `--presplash-lottie`: use a lottie (json) file as a presplash animation. If used, this will replace the static presplash image.
- `--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.
- `--enable-androidx`: Enable AndroidX support library.

webview

You can use this with `--bootstrap=webview`, or include the `webviewjni` recipe, e.g. `--requirements=webviewjni,python3`.

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. 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.

service_library

You can use this with `--bootstrap=service_library` option.

This bootstrap can be used together with `aar` output target to generate a library, containing Python services that can be used with other build systems and frameworks.

- `--private`: The directory containing your project files.
- `--package`: The Java package name for your project. e.g. `org.example.yourapp`.
- `--name`: The library name.
- `--version`: The version number.
- `--service`: A service name and the Python script it should run. See *Arbitrary service scripts*.
- `--blacklist`: The path to a file containing blacklisted patterns that will be excluded from the final AAR. Defaults to `./blacklist.txt`.
- `--whitelist`: The path to a file containing whitelisted patterns that will be included in the AAR 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.
- `add-source`: Add a source directory to the app's Java code.

1.2.3 Requirements blacklist (APK size optimization)

To optimize the size of the *.apk* file that p4a builds for you, you can **blacklist** certain core components. Per default, p4a will add python *with batteries included* as would be expected on desktop, including openssl, sqlite3 and other components you may not use.

To blacklist an item, specify the `--blacklist-requirements` option:

```
p4a apk ... --blacklist-requirements=sqlite3
```

At the moment, the following core components can be blacklisted (if you don't want to use them) to decrease APK size:

- `android` disables p4a's android module (see *android for Android API access*)
- `libffi` disables ctypes stdlib module
- `openssl` disables ssl stdlib module
- `sqlite3` disables sqlite3 stdlib module

1.3 Commands

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

1.3.1 Commands index

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

1.3.2 General arguments

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

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

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

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

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

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

1.3.3 Distribution arguments

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

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

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

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

--force-build BOOL Whether the distribution must be compiled from scratch.

--arch The architecture to build for. You can specify multiple architectures to build for at the same time. As an example `p4a ... --arch arm64-v8a --arch armeabi-v7a ...` will build a distribution for both `arm64-v8a` and `armeabi-v7a`.

--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` (used with Kivy and most other apps) or `webview`.

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

1.4 Working on Android

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

1.4.1 Storage paths

If you want to store and retrieve data, you shouldn't just save to the current directory, and not hardcode */sdcard/* or some other path either - it might differ per device.

Instead, the *android* module which you can add to your *–requirements* allows you to query the most commonly required paths:

```
from android.storage import app_storage_path
settings_path = app_storage_path()
```

(continues on next page)

(continued from previous page)

```
from android.storage import primary_external_storage_path
primary_ext_storage = primary_external_storage_path()

from android.storage import secondary_external_storage_path
secondary_ext_storage = secondary_external_storage_path()
```

app_storage_path() gives you Android's so-called "internal storage" which is specific to your app and cannot seen by others or the user. It compares best to the AppData directory on Windows.

primary_external_storage_path() returns Android's so-called "primary external storage", often found at */sdcard/* and potentially accessible to any other app. It compares best to the Documents directory on Windows. Requires *Permission.WRITE_EXTERNAL_STORAGE* to read and write to.

secondary_external_storage_path() returns Android's so-called "secondary external storage", often found at */storage/External_SD/*. It compares best to an external disk plugged to a Desktop PC, and can after a device restart become inaccessible if removed. Requires *Permission.WRITE_EXTERNAL_STORAGE* to read and write to.

> **Warning:** Even if *secondary_external_storage_path* returns a path the external sd card may still not be present. Only non-empty contents or a successful write indicate that it is.

Read more on all the different storage types and what to use them for in the Android documentation:

https://developer.android.com/training/data-storage/files

A note on permissions

Only the internal storage is always accessible with no additional permissions. For both primary and secondary external storage, you need to obtain *Permission.WRITE_EXTERNAL_STORAGE* **and the user may deny it.** Also, if you get it, both forms of external storage may only allow your app to write to the common pre-existing folders like "Music", "Documents", and so on. (see the Android Docs linked above for details)

1.4.2 Runtime permissions

With API level >= 21, you will need to request runtime permissions to access the SD card, the camera, and other things.

This can be done through the *android* module which is *available per default* unless you blacklist it. Use it in your app like this:

```
from android.permissions import request_permissions, Permission
request_permissions([Permission.WRITE_EXTERNAL_STORAGE])
```

The available permissions are listed here:

https://developer.android.com/reference/android/Manifest.permission

1.4.3 Other common tasks

Dismissing the splash screen

With the SDL2 bootstrap, the app's splash screen may be visible longer than necessary (with your app already being loaded) 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.

To dismiss the loading screen explicitly in your code, use the *android* module:

```
from android import loadingscreen
loadingscreen.hide_loading_screen()
```

You can call it e.g. using `kivy.clock.Clock.schedule_once` to run it in the first active frame of your app, or use the app build 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.).

Observing Activity result

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)
    # ...
```

Receiving Broadcast message

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()
```

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):
```

(continues on next page)

(continued from previous page)

```
    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.

1.4.4 Advanced Android API use

android for Android API access

As mentioned above, the `android` Python module provides a simple wrapper around many native Android APIS, and it is *included by default* unless you blacklist it.

The available functionality of this module is not separately documented. You can read the source on Github.

Also please note you can replicate most functionality without it using *pyjnius*. (see below)

Plyer - a more comprehensive API wrapper

Plyer provides a more thorough wrapper than *android* for a much larger area of platform-specific APIs, supporting not only Android but also 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 use 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!

Pyjnius - raw lowlevel API access

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.

This is particularly useful when *android* and *plyer* don't already provide a convenient access to the API, or you need more control.

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 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 the bootstraps store in the `mActivity` static variable. This difference isn't always important, but it's important to know about.
- The code closely follows the Java API - this is exactly the same set of function calls that you'd use to achieve the same thing from Java code.
- This is quite verbose - it's a lot of lines to achieve a simple vibration!

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

You can check the Pyjnius documentation for further details.

1.5 Launcher

The Kivy Launcher is an Android application that can run any Kivy app stored in the *kivy* folder on the 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.

1.5.1 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 a 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=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.

1.5.2 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` in the folder. The icon should be a square.

1.5.3 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.

1.5.4 Source code

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

1.6 distutils/setuptools integration

1.6.1 Have *p4a apk* run setup.py (replaces `--requirements`)

If your project has a *setup.py* file, then it can be executed by *p4a* when your app is packaged such that your app properly ends up in the packaged site-packages. (Use `--use-setup-py` to enable this, `--ignore-setup-py` to prevent it)

This is functionality to run **setup.py INSIDE 'p4a apk',** as opposed to the other section below, which is about running *p4a inside setup.py*.

This however has these caveats:

- **Only your ''main.py'' from your app's ''–private'' data is copied into the .apk!** Everything else needs to be installed by your `setup.py` into the site-packages, or it won't be packaged.
- All dependencies that map to recipes can only be pinned to exact versions, all other constraints will either just plain not work or even cause build errors. (Sorry, our internal processing is just not smart enough to honor them properly at this point)
- The dependency analysis at the start may be quite slow and delay your build

Reasons why you would want to use a *setup.py* to be processed (and omit specifying `--requirements`):

- You want to use a more standard mechanism to specify dependencies instead of `--requirements`
- You already use a *setup.py* for other platforms
- Your application imports itself in a way that won't work unless installed to site-packages)

Reasons **not** to use a *setup.py* (that is to use the usual `--requirements` mechanism instead):

- You don't use a *setup.py* yet, and prefer the simplicity of just specifying `--requirements`
- Your *setup.py* assumes a desktop platform and pulls in Android-incompatible dependencies, and you are not willing to change this, or you want to keep it separate from Android deployment for other organizational reasons
- You need data files to be around that aren't installed by your *setup.py* into the site-packages folder

1.6.2 Use your setup.py to call p4a

Instead of running p4a via the command line, you can call it via *setup.py* instead, by it integrating with distutils and setup.py.

This is functionality to run **p4a INSIDE setup.py,** as opposed to the other section above, which is about running *setup.py inside 'p4a apk'*.

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 setuptools import find_packages

setup(
    name='testapp_setup',
    version='1.1',
```

(continues on next page)

(continued from previous page)

```
    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,python3',
                   '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
```

1.7 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 for Python modules which have no compiled components; these are installed automatically 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.

1.7.1 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 = ['genericknkbuild']

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().prebuild_arch(arch)
    # Do any pre-initialisation

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

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

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

If you defined a 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().build_arch(arch)
    with current_directory(self.get_build_dir(arch.arch)):
        with open('example_file.txt', 'w') as fileh:
            fileh.write('This is written to a file within the build dir')
```

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

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

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

1.7.2 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().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 use in 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().get_recipe_env(arch)
    env['USE_SDL2'] = '1'
```

(continues on next page)

(continued from previous page)

```
    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
```

1.7.3 Using a PythonRecipe

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

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

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

    depends = ['python3', '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().build_arch(arch)
    self.install_python_package()

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

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

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

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

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

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

1.7.4 Using a CythonRecipe

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

For instance, the following is all that's necessary to make a recipe for Kivy:

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

    depends = ['sdl2', 'pyjnius']

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)
```

(continues on next page)

(continued from previous page)

```
    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, firstly to make sure that any .pyx files have been generated by setup.py, and secondly because cython isn't installed in the hostpython build.

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

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

1.7.5 Using a CompiledComponentsPythonRecipe

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

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

1.7.6 Using an NDKRecipe

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

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

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

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

recipe = LibSDL2TTF()
```

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

1.7.7 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 = ['python3', '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().get_recipe_env(arch)
        # Manipulate the env here if you want
        return env

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

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

(continues on next page)

(continued from previous page)

```
            sh.ls('-lathr')  # Or run some commands that actually do
                             # something

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


recipe = YourRecipe()
```

1.7.8 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.

1.7.9 The `Recipe` class

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

1.8 Bootstraps

This page is about creating new bootstrap backends. For build options of existing bootstraps (i.e. with SDL2, 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 projects for Android.

1.8.1 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'
```

(continues on next page)

(continued from previous page)

```
    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.

1.9 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. Please note that python-for-android explicitly runs services as separated processes by having a colon ":" in the beginning of the name assigned to the `android:process` attribute of the `AndroidManifest.xml` file. This is not the default behavior, see Android service documentation. You can communicate with the service process from your app using e.g. osc or (a heavier option) twisted.

1.9.1 Service creation

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

Service folder

This is the older method of handling services. 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 (you may need to add `android` to your app requirements):

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

Arbitrary service scripts

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_SERVICE_PY` argument when calling python-for-android, or in buildozer.spec, a `services = myservice:PATH_TO_SERVICE_PY` [app] setting.

The `myservice` name before the colon is the name of the service class, via which you will interact with it later.

The `PATH_TO_SERVICE_PY` is the relative path to the service entry point (like `services/myservice.py`)

You can optionally specify the following parameters:

- `:foreground` for launching a service as an Android foreground service
- `:sticky` for launching a service that gets restarted by the Android OS on exit/error

Full command with all the optional parameters included would be: `--service=myservice:services/myservice.py:foreground:sticky`

You can add multiple `--service` arguments to include multiple services, or separate them with a comma in buildozer.spec, all of 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.domain.package.name.ServiceMyservice')
mActivity = autoclass('org.kivy.android.PythonActivity').mActivity
argument = ''
service.start(mActivity, argument)
```

Here, `your.package.domain.package.name` refers to the package identifier of your APK.

If you are using buildozer, the identifier is set by the `package.name` and `package.domain` values in your buildozer.spec file. The name of the service is `ServiceMyservice`, where `Myservice` is the name specied by one of the `services` values, but with the first letter upper case.

If you are using python-for-android directly, the identifier is set by the `--package` argument to python-for-android. The name of the service is `ServiceMyservice`, where `Myservice` is the identifier that was previously passed to the `--service` argument, but 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.

The service argument is made available to your service via the 'PYTHON_SERVICE_ARGUMENT' environment variable. It is exposed as a simple string, so if you want to pass in multiple values, we would recommend using the json module to encode and decode more complex data.

```
from os import environ
argument = environ.get('PYTHON_SERVICE_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.

Service auto-restart

It is possible to make services restart automatically when they exit by calling `setAutoRestartService(True)` on the service object. The call to this method should be done within the service code:

```
from jnius import autoclass
PythonService = autoclass('org.kivy.android.PythonService')
PythonService.mService.setAutoRestartService(True)
```

1.10 Troubleshooting

1.10.1 Debug output

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

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

1.10.2 Getting help

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

- by email to the kivy-users Google group
- on #support Discord channel

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

1.10.3 Debugging on Android

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

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

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

or:

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

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

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

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

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

1.10.4 Unpacking an APK

It is sometimes useful to unpack a packaged 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 user app data (code, images, fonts ..) is packaged into a single tarball contained in the assets folder:

```
$ cd assets
$ ls
private.tar
```

`private.tar` is a tarball containing all your packaged data. Extract it:

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

This will reveal all the user app data (the files shown below are from the touchtracer demo):

```
$ ls
README.txt            android.txt             icon.png                main.pyc
↪        p4a_env_vars.txt        particle.png
private.tar           touchtracer.kv
```

Due to how We're required to ship ABI-specific things in Android App Bundle, the Python installation is packaged separately, as (most of it) is ABI-specific.

For example, the Python installation for `arm64-v8a` is available in `lib/arm64-v8a/libpybundle.so`

`libpybundle.so` is a tarball (but named like a library for packaging requirements), that contains our `_python_bundle`:

```
$ tar xf libpybundle.so
$ cd _python_bundle
$ ls
modules               site-packages   stdlib.zip
```

1.10.5 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=python3,kivy`. This also applies when using buildozer, in which case add python3 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.

Requested API target 19 is not available, install it with the SDK android tool

This means that your SDK is missing the required platform tools. You need to install the `platforms;android-19` package in your SDK, using the `android` or `sdkmanager` tools (depending on SDK version).

If using buildozer this should be done automatically, but as a workaround you can run these from `~/.buildozer/android/platform/android-sdk-20/tools/android`.

ModuleNotFoundError: No module named '_ctypes'

You do not have the libffi headers available to python-for-android, so you need to install them. On Ubuntu and derivatives these come from the *libffi-dev* package.

After installing the headers, clean the build (*p4a clean builds*, or with buildozer delete the *.buildozer* directory within your app directory) and run python-for-android again.

SSLError("Can't connect to HTTPS URL because the SSL module is not available.")

Your *hostpython3* was compiled without SSL support. You need to install the SSL development files before rebuilding the *hostpython3* recipe. Remember to always clean the build before rebuilding (*p4a clean builds*, or with buildozer *buildozer android clean*).

On Ubuntu and derivatives:

```
apt install libssl-dev
p4a clean builds # or with: buildozer `buildozer android clean
```

On macOS:

```
brew install openssl
sudo ln -sfn /usr/local/opt/openssl /usr/local/ssl
p4a clean builds # or with: buildozer `buildozer android clean
```

1.11 Docker

Currently we use a containerized build for testing Python for Android recipes. Docker supports three big platforms either directly with the kernel or via using headless VirtualBox and a small distro to run itself on.

While this is not the actively supported way to build applications, if you are willing to play with the approach, you can use the `Dockerfile` to build the Docker image we use for CI builds and create an Android application with that in a container. This approach allows you to build Android applications on all platforms Docker engine supports. These steps assume you already have Docker preinstalled and set up.

Warning: This approach is highly space unfriendly! The more layers (`commit`) or even Docker images (`build`) you create the more space it'll consume. Within the Docker image there is Android SDK and NDK + various dependencies. Within the custom diff made by building the distribution there is another big chunk of space eaten. The very basic stuff such as a distribution with: CPython 3, setuptools, Python for Android `android` module, SDL2 (+ deps), PyJNIus and Kivy takes almost 2 GB. Check your free space first!

1. Clone the repository:

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

2. Build the image with name `p4a`:

```
docker build --tag p4a .
```

Note: You need to be in the `python-for-android` for the Docker build context and you can optionally use `--file` flag to specify the path to the `Dockerfile` location.

3. Create a container from `p4a` image with copied `testapps` folder in the image mounted to the same one in the cloned repo on the host:

```
docker run \
    --interactive \
    --tty \
    --volume ".../testapps":/home/user/testapps \
    p4a sh -c
        '. venv/bin/activate \
        && cd testapps \
        && python setup_testapp_python3.py apk \
        --sdk-dir $ANDROID_SDK_HOME \
        --ndk-dir $ANDROID_NDK_HOME'
```

Note: On Windows you might need to use quotes and forward-slash path for volume "/c/Users/. . . /python-for-android/testapps":/home/user/testapps

Warning: On Windows `gradlew` will attempt to use 'bashr' command which is a result of Windows line endings. For that you'll need to install `dos2unix` package into the image.

4. Preserve the distribution you've already built (optional, but recommended):

   docker commit $(docker ps –last=1 –quiet) my_p4a_dist

5. Find the `.APK` file on this location:

```
ls -lah testapps
```

1.12 Development and 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.

Read on for more information about how we manage development and releases, but don't worry about the details! Pull requests are welcome and we'll deal with the rest.

1.12.1 Development model

python-for-android is developed using the following model:

- The `master` branch always represents the latest stable release.
- The `develop` branch is the most up to date with new contributions.
- Releases happen periodically, and consist of merging the current `develop` branch into `master`.

For reference, this is based on a Git flow model, although we don't follow this religiously.

1.12.2 Versioning

python-for-android releases currently use calendar versioning. Release numbers are of the form YYYY.MM.DD. We aim to create a new release every four weeks, but more frequent releases are also possible.

We use calendar versioning because in practice, changes in python-for-android are often driven by updates or adjustments in the Android build tools. It's usually best for users to be working from the latest release. We try to maintain backwards compatibility even while internals are changing.

1.12.3 Creating a new release

New releases follow these steps:

- Create a new branch `release-YYYY.MM.DD` based on the `develop` branch. - `git checkout -b release-YYYY.MM.DD develop`
- Create a Github pull request to merge `release-YYYY.MM.DD` into `master`.
- Complete all steps in the *release checklist*, and document this in the pull request (copy the checklist into the PR text)

At this point, wait for reviewer approval and conclude any discussion that arises. To complete the release:

- Merge the release branch to the `master` branch.
- Also merge the release branch to the `develop` branch.
- Tag the release commit in `master`, with tag `vYYYY.MM.DD`. Include a short summary of the changes.
- Release distributions and PyPI upload should be handled by the CI.
- Add to the Github release page (see e.g. this example): - The python-for-android README summary - A short list of major changes in this release, if any - A changelog summarising merge commits since the last release - The release sdist and wheel(s)

Release checklist

```
- [ ] Check that the builds are passing
  - [ ] [GitHub Action](https://github.com/kivy/python-for-android/actions)
- [ ] Run the tests locally via `tox`: this performs some long-running tests that are␣
↪skipped on github-actions.
- [ ] Build and run the [on_device_unit_tests](https://github.com/kivy/python-for-
↪android/tree/master/testapps/on_device_unit_tests) app using buildozer. Check that␣
↪they all pass.
- [ ] Build (or download from github actions) and run the following [testapps](https:/
↪/github.com/kivy/python-for-android/tree/master/testapps/on_device_unit_tests) for␣
↪arch `armeabi-v7a` and `arm64-v8a`:
  - [ ] on_device_unit_tests
    - [ ] `armeabi-v7a` (`cd testapps/on_device_unit_tests && PYTHONPATH=.:../../␣
↪python3 setup.py apk  --ndk-dir=<your-ndk-dir> --sdk-dir=<your-sdk-dir> --
↪arch=armeabi-v7a --debug`)
    - [ ] `arm64-v8a` (`cd testapps/on_device_unit_tests && PYTHONPATH=.:../../␣
↪python3 setup.py apk  --ndk-dir=<your-ndk-dir> --sdk-dir=<your-sdk-dir> --
↪arch=arm64-v8a --debug`)
- [ ] Check that the version number is correct
```

1.12.4 How python-for-android uses *pip*

Last update: July 2019

This section is meant to provide a quick summary how p4a (=python-for-android) uses pip and python packages in its build process. **It is written for a python packagers point of view, not for regular end users or contributors,** to assist with making pip developers and other packaging experts aware of p4a's packaging needs.

Please note this section just attempts to neutrally list the current mechanisms, so some of this isn't necessarily meant to stay but just how things work inside p4a in this very moment.

Basic concepts

(This part repeats other parts of the docs, for the sake of making this a more independent read)

p4a builds & packages a python application for use on Android. It does this by providing a Java wrapper, and for graphical applications an SDL2-based wrapper which can be used with the kivy UI toolkit if desired (or alternatively just plain PySDL2). Any such python application will of course have further library dependencies to do its work.

p4a supports two types of package dependencies for a project:

Recipe: install script in custom p4a format. Can either install C/C++ or other things that cannot be pulled in via pip, or things that can be installed via pip but break on android by default. These are maintained primarily inside the p4a source tree by p4a contributors and interested folks.

Python package: any random pip python package can be directly installed if it doesn't need adjustments to work for Android.

p4a will map any dependency to an internal recipe if present, and otherwise use pip to obtain it regularly from whatever external source.

Install process regarding packages

The install/build process of a p4a project, as triggered by the *p4a apk* command, roughly works as follows in regards to python packages:

1. The user has specified a project folder to install. This is either just a folder with python scripts and a *main.py*, or it may also have a *pyproject.toml* for a more standardized install.
2. Dependencies are collected: they can be either specified via `--requirements` as a list of names or pip-style URLs, or p4a can optionally scan them from a project folder via the pep517 library (if there is a *pyproject.toml* or *setup.py*).
3. The collected dependencies are mapped to p4a's recipes if any are available for them, otherwise they're kept around as external regular package references.
4. All the dependencies mapped to recipes are built via p4a's internal mechanisms to build these recipes. (This may or may not indirectly use pip, depending on whether the recipe wraps a python package or not and uses pip to install or not.)
5. **If the user has specified to install the project in standardized ways,** then the *setup.py*/whatever build system of the project will be run. This happens with cross compilation set up (*CC*/*CFLAGS*/... set to use the proper toolchain) and a custom site-packages location. The actual comand is a simple *pip install .* in the project folder with some extra options: e.g. all dependencies that were already installed by recipes will be pinned with a *-c* constraints file to make sure pip won't install them, and build isolation will be disabled via `--no-build-isolation` so pip doesn't reinstall recipe-packages on its own.

   If the user has not specified to use standardized build approaches, p4a will simply install all the remaining dependencies that weren't mapped to recipes directly and just plain copy in the user project without installing. Any *setup.py* or *pyproject.toml* of the user project will then be ignored in this step.
6. Google's gradle is invoked to package it all up into an *.apk*.

Overall process / package relevant notes for p4a

Here are some common things worth knowing about python-for-android's dealing with python packages:

- Packages will work fine without a recipe if they would also build on Linux ARM, don't use any API not available in the NDK if they use native code, and don't use any weird compiler flags the toolchain doesn't like if they use native code. The package also needs to work with cross compilation.
- There is currently no easy way for a package to know it is being cross-compiled (at least that we know of) other than examining the *CC* compiler that was set, or that it is being cross-compiled for Android specifically. If that breaks a package it currently needs to be worked around with a recipe.
- If a package does **not** work, p4a developers will often create a recipe instead of getting upstream to fix it because p4a simply is too niche.
- Most packages without native code will just work out of the box. Many with native code tend not to, especially if complex, e.g. numpy.
- Anything mapped to a p4a recipe cannot be just reinstalled by pip, specifically also not inside build isolation as a dependency. (It *may* work if the patches of the recipe are just relevant to fix runtime issues.) Therefore as of now, the best way to deal with this limitation seems to be to keep build isolation always off.

Ideas for the future regarding packaging

- We in overall prefer to use the recipe mechanism less if we can. In overall the recipes are just a collection of workarounds. It may look quite hacky from the outside, since p4a version pins recipe-wrapped packages usually to make the patches reliably apply. This creates work for the recipes to be kept up-to-date, and obviously this approach doesn't scale too well. However, it has ended up as a quite practical interims solution until better ways are found.
- Obviously, it would be nice if packages could know they are being cross-compiled, and for Android specifically. We aren't currently aware of a good mechanism for that.

- If pip could actually run the recipes (instead of p4a wrapping pip and doing so) then this might even allow build isolation to work - but this might be too complex to get working. It might be more practical to just gradually reduce the reliance on recipes instead and make more packages work out of the box. This has been done e.g. with improvements to the cross-compile environment being set up automatically, and we're open for any ideas on how to improve this.

1.13 Testing an python-for-android pull request

In order to test a pull request, we recommend to consider the following points:

1. of course, check if the overall thing makes sense
2. is the CI passing? if not what specifically fails
3. is it working locally at compile time?
4. is it working on device at runtime?

This document will focus on the third point: *is it working locally at compile time?* so we will give some hints about how to proceed in order to create a local copy of the pull requests and build an apk. We expect that the contributors has enough criteria/knowledge to perform the other steps mentioned, so let's begin...

To create an apk from a python-for-android pull request we contemplate three possible scenarios:

- using python-for-android commands directly from the pull request files that we want to test, without installing it (the recommended way for most of the test cases)
- installing python-for-android using the github's branch of the pull request
- using buildozer and a custom app

We will explain the first two methods using one of the distributed python-for-android test apps and we assume that you already have the python-for-android dependencies installed. For the *buildozer* method we also expect that you already have a a properly working app to test and a working installation/configuration of *buildozer*. There is one step that it's shared with all the testing methods that we propose in here... we named it *Common steps*.

1.13.1 Common steps

The first step to do it's to get a copy of the pull request, we can do it of several ways, and that it will depend of the circumstances but all the methods presented here will do the job, so...

Fetch the pull request by number

For the example, we will use *1901* for the example) and the pull request branch that we will use is *feature-fix-numpy*, then you will use a variation of the following git command: *git fetch origin pull/<#>/head:<local_branch_name>*, e.g.:

```
git fetch upstream pull/1901/head:feature-fix-numpy
```

Note: Notice that we fetch from *upstream*, since that is the original project, where the pull request is supposed to be

Tip: The amount of work of some users maybe worth it to add his remote to your fork's git configuration, to do so with the imaginary github user *Obi-Wan Kenobi* which nickname is *obiwankenobi*, you will do:

```
git remote add obiwankenobi https://github.com/obiwankenobi/python-for-
↪android.git
```

And to fetch the pull request branch that we put as example, you would do:

```
git fetch obiwankenobi
git checkout obiwankenobi/feature-fix-numpy
```

Clone the pull request branch from the user's fork

Sometimes you may prefer to use directly the fork of the user, so you will get the nickname of the user who created the pull request, let's take the same imaginary user than before *obiwankenobi*:

```
git clone -b feature-fix-numpy \
    --single-branch \
    https://github.com/obiwankenobi/python-for-android.git \
    p4a-feature-fix-numpy
```

Here's the above command explained line by line:

- *git clone -b feature-fix-numpy*: we tell git that we want to clone the branch named *feature-fix-numpy*
- *–single-branch*: we tell git that we only want that branch
- *https://github.com/obiwankenobi/python-for-android.git*: noticed the nickname of the user that created the pull request: *obiwankenobi* in the middle of the line? that should be changed as needed for each pull request that you want to test
- *p4a-feature-fix-numpy*: the name of the cloned repository, so we can have multiple clones of different prs in the same folder

Note: You can view the author/branch information looking at the subtitle of the pull request, near the pull request status (expected an *open* status)

1.13.2 Using python-for-android commands directly from the pull request files

- Enter inside the directory of the cloned repository in the above step and run p4a command with proper args, e.g. (to test an modified *pycryptodome* recipe)

```
cd p4a-feature-fix-numpy
PYTHONPATH=. python3 -m pythonforandroid.toolchain apk \
    --private=testapps/on_device_unit_tests/test_app \
    --dist-name=dist_unit_tests_app_pycryptodome \
    --package=org.kivy \
    --name=unit_tests_app_pycryptodome \
    --version=0.1 \
    --requirements=sdl2,pyjnius,kivy,python3,pycryptodome \
    --ndk-dir=/media/DEVEL/Android/android-ndk-r20 \
    --sdk-dir=/media/DEVEL/Android/android-sdk-linux \
    --android-api=27 \
    --arch=arm64-v8a \
    --permission=VIBRATE \
    --debug
```

Things that you should know:

- The example above will build an test app we will make use of the files of the *on device unit tests* test app but we don't use the setup file to build it so we must tell python-for-android what we want via arguments
- be sure to at least edit the following arguments when running the above command, since the default set in there it's unlikely that match your installation:
  - *–ndk-dir*: An absolute path to your android's NDK dir
  - *–sdk-dir*: An absolute path to your android's SDK dir
  - *–debug*: this one enables the debug mode of python-for-android, which will show all log messages of the build. You can omit this one but it's worth it to be mentioned, since this it's useful to us when trying to find the source of the problem when things goes wrong
- The apk generated by the above command should be located at the root of of the cloned repository, were you run the command to build the apk
- The testapps distributed with python-for-android are located at *testapps* folder under the main folder project
- All the builds of python-for-android are located at *~/.local/share/python-for-android*
- You should have a downloaded copy of the android's NDK and SDK

1.13.3 Installing python-for-android using the github's branch of the pull request

- Enter inside the directory of the cloned repository mentioned in *Common steps* and install it via pip, e.g.:

```
cd p4a-feature-fix-numpy
pip3 install . --upgrade --user
```

- Now, go inside the *testapps/on_device_unit_tests* directory (we assume that you still are inside the cloned repository)

```
cd testapps/on_device_unit_tests
```

- Run the build of the apk via the freshly installed copy of python-for-android by running a similar command than below

```
python3 setup.py apk \
    --ndk-dir=/media/DEVEL/Android/android-ndk-r20 \
    --sdk-dir=/media/DEVEL/Android/android-sdk-linux \
    --android-api=27 \
    --arch=arm64-v8a \
    --debug
```

Things that you should know:

- In the example above, we override some variables that are set in *setup.py*, you could also override them by editing this file
- be sure to at least edit the following arguments when running the above command, since the default set in there it's unlikely that match your installation:
  - *–ndk-dir*: An absolute path to your android's NDK dir
  - *–sdk-dir*: An absolute path to your android's SDK dir

Tip: if you don't want to mess up with the system's python, you could do the same steps but inside a virtualenv

Warning: Once you finish the pull request tests remember to go back to the master or develop versions of python-for-android, since you just installed the python-for-android files of the *pull request*

1.13.4 Using buildozer with a custom app

- Edit your *buildozer.spec* file. You should search for the key *p4a.source_dir* and set the right value so in the example posted in *Common steps* it would look like this:

```
p4a.source_dir = /home/user/p4a_pull_requests/p4a-feature-fix-numpy
```

- Run you buildozer command as usual, e.g.:

```
buildozer android debug p4a --dist-name=dist-test-feature-fix-numpy
```

Note: this method has the advantage, can be run without installing the pull request version of python-for-android nor the android's dependencies but has one problem... when things goes wrong you must determine if it's a buildozer issue or a python-for-android one

Warning: Once you finish the pull request tests remember to comment/edit the *p4a.source_dir* constant that you just edited to test the pull request

Tip: this method it's useful for developing pull requests since you can edit *p4a.source_dir* to point to your python-for-android fork and you can test any branch you want only switching branches with: *git checkout <branch-name>* from inside your python-for-android fork

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

S

U