BUILD PYTHON BASED ANDROID/iOS APPS USING KIVY (CROSSPLATFORM APPS)

Author: Himanshu Bobade

Abstract

Over-time smartphone software platforms having grown rapidly and it has become difficult for developers to keep track of different set of technologies. A solution to this is python kivy framework. For a python user, Kivy is easy, reliable and helps one to design innovative user interfaces with multi-touch functionalities. This research paper demonstrates basics of kivy widgets, layouts, kv language and android packaging.

Introduction

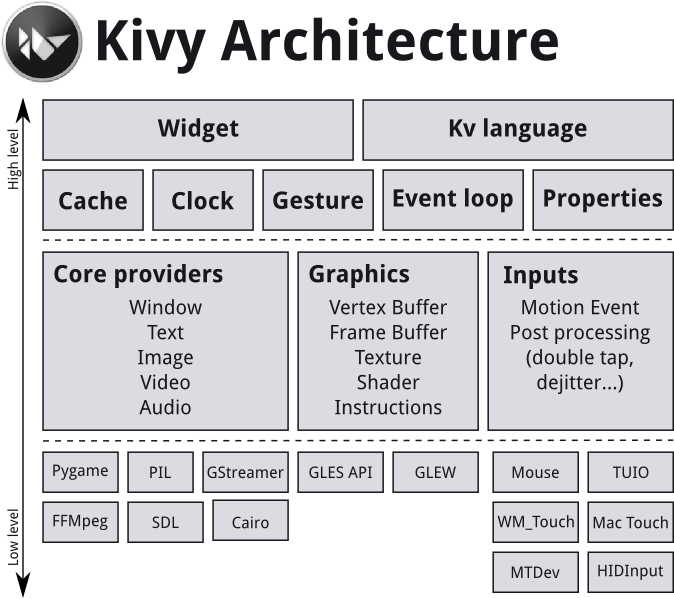
The modern smartphone has taken a long 29-year journey to reach us in 2021, and it's changed a lot along the way. It’s an evolution that's taken the market by storm and in these past years not only smartphones have grown but also the seeds of Operating system platforms have cultivated massively. When a company develops an application, it often faces the challenge of needing to develop it for multiple platforms. Android has been a leading mobile operating system worldwide for last 5 years, controlling the mobile OS market with 50-70% share. Apple’s iOS and Google’s Android together possess over 99% of the global market share. A company will have to hire a set of two development teams to develop a same application. This increases the cost of the project and is very reluctant when it comes to regular app updates. A solution to this is Cross-Platform app development.

Cross-platform app developers create a unified API running on top of a native SDK, make use of native IDEs, and build iOS and Android apps that share the same codebase. Various frameworks like IONIC, Facebook’s React Native, Google’s Flutter exerts developers to learn JavaScript, Java, Kotlin, Dart, etc. and thus to develop an app you need to learn these languages separately. Python has gained popularity over years and is well favoured by developers. According to SlashData, there are now 8.2 million developers in the world who code using Python. It would be reliable if there was a way to use python and develop apps directly. It is possible and can be done by using python kivy framework.

Kivy is a free and open source Python framework for developing mobile apps and other multi-touch application software with a natural user interface. It is distributed under the terms of the MIT License, and is platform independent that is it can intrinsically run on platforms like Android, iOS, Meego, OSX , Raspberry Pi, Linux , and Windows. The graphics engine is built over OpenGL ES 2, using a modern and fast graphics pipeline.

Architecture

The kivy architecture is illustrated below in fig 1.0



**Fig 1.0 Architecture of kivy**

Core Providers and Input Providers

Modularity and abstraction are the most important internals of kivy. Abstraction has a key role when comes about basic but core tasks such as opening a window, displaying images and text, playing audio, getting images from a camera, spelling correction and so on making it easier for the API to use and extend. One of the key advantages of using specialized core providers for each platform is that it can fully leverage the functionality exposed by the operating system and act as efficiently as possible. By using libraries that are shipped with any one platform, efficiently reduces the size of the Kivy distribution and make packaging easier. This also makes it easier to port Kivy to other platforms especially Android.

Graphics

As mentioned earlier, abstraction has an important role in the architecture of kivy. Kivy’s graphics API is an abstraction of OpenGL. For developer’s reliance, kivy provides simple metaphors that do not exist as such in OpenGL (e.g. Canvas, Rectangle, etc.).

One of the advantages of the graphics API is its ability to automatically optimize the drawing commands that the code issues. This makes the drawing code more efficient in many cases. Kivy uses OpenGL version 2.0 ES (GLES2)

Core

The code in the core package provides commonly used features, such as:

Clock: Used to schedule timer events.

Cache: Used to cache something that is often used.

Gesture Detection: Used to detect different kinds of strokes such as circles or rectangles, shipped with simple gesture recognizer.

Kivy Language: The kivy language is used to easily and describe user interfaces. It is used more for styling purposes.

Properties: Used to link the widget code with the user interface description.

Create an application

Creating an App with kivy is really very simple. A sub-classing parameter should be used to define App. Initiate a simple build() method which will return a Widget instance. Now the class defined should be instantiated just like it is done in the python traditionally with another inbuilt function called run().

Here is an example of a minimal application:

from kivy.app import App

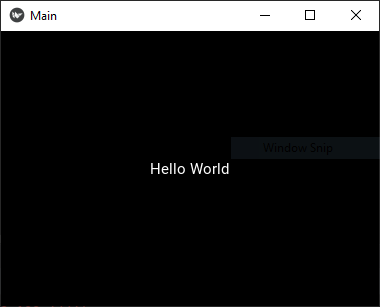
class MainApp(App):

def build(self):

return

MainApp().run()

The piece of code will generate a Kivy App window and will be blank as no parameter was passed into the kivy App.



**Fig 2.0** Executed App(with a label “Hello world”) using run function

If a simple label is added to the display a text on the App, the output screen will look like (Essential label libraries need to be imported) the one in fig 2.0:

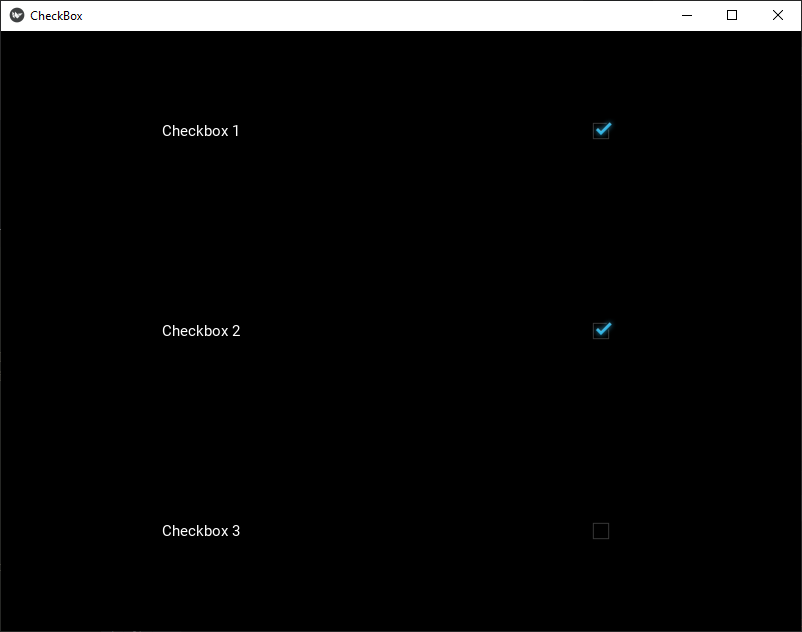
UIX (Widgets & Layouts)

Almost everything one can see on the App screen is a widget! Widgets and layouts are used to design the user iinterface. A widget is a user interface element that is added to serve some kind of functionality. For eg. Lists, sliders, Buttons, etc. Motion-events are acted upon a widget. Layouts are a way to arrange the widgets.

Widgets Z Index: The order of widget drawing is considered on the basis of widget’s position in the widget tree. The lower indexed widgets will be drawn above those with a higher index. The default for index is 0.

Kivy widgets can be categorized as follows:

**UX widgets**: Classical user interface widgets, ready to be assembled to create more complex widgets. Eg: [Label](https://kivy.org/doc/stable/api-kivy.uix.label.html)(Fig 2.0), [Button](https://kivy.org/doc/stable/api-kivy.uix.button.html)(Fig 3.0), [CheckBox](https://kivy.org/doc/stable/api-kivy.uix.checkbox.html)(Fig 4.0), [Image](https://kivy.org/doc/stable/api-kivy.uix.image.html)(Fig 3.0), [Slider](https://kivy.org/doc/stable/api-kivy.uix.slider.html), [Progress Bar](https://kivy.org/doc/stable/api-kivy.uix.progressbar.html), [Text Input](https://kivy.org/doc/stable/api-kivy.uix.textinput.html), [Toggle button](https://kivy.org/doc/stable/api-kivy.uix.togglebutton.html), [Switch](https://kivy.org/doc/stable/api-kivy.uix.switch.html), [Video](https://kivy.org/doc/stable/api-kivy.uix.video.html).



**Fig 3.0** (Image and Button widget) **Fig 4.0** (CheckBox)

Image, Button and checkbox is demonstrated above.

**Layouts**: A layout widget does no rendering but just acts as a trigger that arranges its children in a specific way. Read more on [Layouts here](https://kivy.org/doc/stable/api-kivy.uix.layout.html). Eg. [Anchor Layout](https://kivy.org/doc/stable/api-kivy.uix.anchorlayout.html), [Box Layout](https://kivy.org/doc/stable/api-kivy.uix.boxlayout.html), [Float Layout](https://kivy.org/doc/stable/api-kivy.uix.floatlayout.html), [Grid Layout](https://kivy.org/doc/stable/api-kivy.uix.gridlayout.html), [PageLayout](https://kivy.org/doc/stable/api-kivy.uix.pagelayout.html), [Relative Layout](https://kivy.org/doc/stable/api-kivy.uix.relativelayout.html), [Scatter Layout](https://kivy.org/doc/stable/api-kivy.uix.scatterlayout.html), [Stack Layout](https://kivy.org/doc/stable/api-kivy.uix.stacklayout.html).

[Layout](https://kivy.org/doc/stable/api-kivy.uix.layout.html#module-kivy.uix.layout) is a special kind of widget that controls the position and size of its children. Layouts use [pos\_hint](https://kivy.org/doc/stable/api-kivy.uix.widget.html" \l "kivy.uix.widget.Widget.pos_hint" \o "kivy.uix.widget.Widget.pos_hint) and [size\_hint](https://kivy.org/doc/stable/api-kivy.uix.widget.html#kivy.uix.widget.Widget.size_hint) properties to determine the position and [size](https://kivy.org/doc/stable/api-kivy.uix.widget.html#kivy.uix.widget.Widget.pos) of their [children](https://kivy.org/doc/stable/api-kivy.uix.widget.html#kivy.uix.widget.Widget.children).

**BoxLayout: Arranges widgets in an adjacent manner (either horizontally or vertically) manner, to fill all the space. Fig 5.0**

**GridLayout: Widgets are in a grid. At least one dimension of the grid must be specified so kivy can compute the size of the elements and how to arrange them. Fig 6.0**

**StackLayout: Arranges widgets adjacent to one another, without trying to make them fit within the entire space but with a set size in one of the dimensions. Fig 7.0**

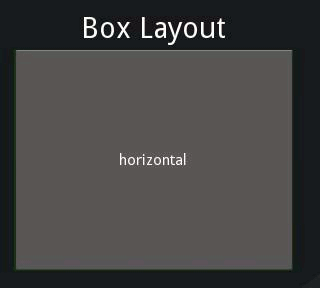
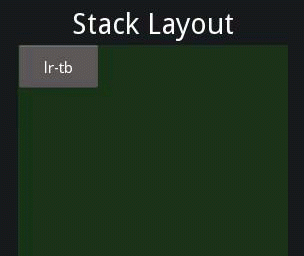
****  

Fig 5.0 (Box Layout) Fig 6.0 (Grid Layout) Fig 7.0 (Stack Layout)

**AnchorLayout: A simple layout which only cares about children positions. It allows putting the children at a position relative to a border of the layout. Fig 8.0**

**FloatLayout: Allows placing children with arbitrary locations and size, either absolute or relative to the layout size. Default size\_hint (1, 1) will make every child the same size as the whole layout, so you probably want to change this value if you have more than one child. You can set size\_hint to (None, None) to use absolute size with size. This widget honors pos\_hint also, which as a dict setting position relative to layout position. Fig 9.0**

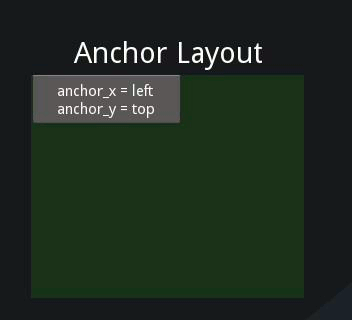


Fig 8.0 ( Anchor Layout) Fig 9.0 (Float Layout)

**RelativeLayout: It is a FloatLayout with an exception that children positions are relative to layout position.**

**Complex UX widgets: Non-atomic widgets that are the result of combining multiple classic widgets. Eg. Bubble, Drop-Down List, FileChooser, Popup, Spinner, RecycleView, TabbedPanel, Videoplayer, VKeyboard.**

**Behaviors widgets**: These widgets do no render but act on the graphics instructions or touch behavior of their children. Eg. [Scatter](https://kivy.org/doc/stable/api-kivy.uix.scatter.html), [Stencil View](https://kivy.org/doc/stable/api-kivy.uix.stencilview.html)

**Screen manager**: Manages screens and transitions when switching from one to another. Eg. [Screen Manager](https://kivy.org/doc/stable/api-kivy.uix.screenmanager.html)

# Kv language

As application grows more complex, it’s common that the construction of widget trees and explicit declaration of bindings becomes hard to maintain. The KV Language is designed to explicitly code the styling for the app. In this way, an organised one can maintain the styling in an organised way. The kivy language allows to create widget tree in a declarative way and to bind widget properties to each other. It allows for very fast prototypes along with agile changes to your UI. It also facilitates separating the logic of the application and its User Interface.

There are two ways to load Kv code into the application:

* By name convention:

Here is a key thing to notice that kivy looks for a Kv file with the same name as your App class in lowercase, minus “App” if it ends with ‘App’ e.g:

MyApp -> my.kv

* Builder: A direct path can be provided to kv file. Also if this string or file defines a root widget, it will be returned by the method:

Builder.load\_file**(**'path/to/file.kv'**)** or Builder.load\_string**(**kv\_string**)**

Kv comes up with certain rules to manage widget content. Kv allows having one root rule, and any number of class or template rules. The root rule is declared by declaring the class of your root widget, followed by “:” and will be set as the root attribute of the App instance. Eg. Widget**:**

A class rule, declared by the name of a widget class between < > and followed by “:” which defines the appearance and behavior of any instance of that class. Eg. <MyWidget>**:**

There are three keywords specific to the Kv language such as app, root, self. There is a special syntax to define values for the whole Kv context. To access Python modules and classes from kv, use #:import. Eg. #:import name x.y.z is equivalent to from x.y import z as name in Python. Similarly, to set a global value, use #:set. Eg. *#:set name value* is equivalent to name=value in Python.

Instantiate children

For declaring a widget instance of some class as a child widget, just declare that child inside the rule:

MyRootWidget**:**

BoxLayout**:**

Button**:**

The example above defines that our root widget, an instance of MyRootWidget, has a child that is an instance of the [BoxLayout](https://kivy.org/doc/stable/api-kivy.uix.boxlayout.html" \l "kivy.uix.boxlayout.BoxLayout" \o "kivy.uix.boxlayout.BoxLayout), and that BoxLayout further has one child that is an instance of the [Button](https://kivy.org/doc/stable/api-kivy.uix.button.html#kivy.uix.button.Button) class.

To specify their behaviour, pass keyword arguments to widgets at creation. For example, to set the number of columns of a [gridlayout](https://kivy.org/doc/stable/api-kivy.uix.gridlayout.html" \l "module-kivy.uix.gridlayout" \o "kivy.uix.gridlayout):

GridLayout**:**

cols**:** 3

Extend canvas

Kv lang can be used to define the canvas instructions in the following way:

MyWidget**:**

canvas**:**

Color**:**

rgba**:** 1**,** .3**,** .8**,** .5

Line**:**

points**:** zip**(**self.data.x**,** self.data.y**)**

To keep the widget alive, a direct reference to the label\_widget widget should be kept. This is achieved using id.\_\_self\_\_ or label\_widget.\_\_self\_\_ in this case. The correct way to do this would be:

<MyWidget>**:**

label\_widget**:** label\_widget.\_\_self\_\_

Designing with the Kivy Language

One of aims of the Kivy language is to [separate the concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) of logic and presentation . The presentation (layout) side is addressed by kv file and the logic by .py file.

Example: a Python file named main.py:

**class** Controller**(**FloatLayout**):**

label\_wid = ObjectProperty**()**

info = StringProperty**()**

**def** do\_action**(**self**):**

self.label\_wid.text = 'My label after button press'

self.info = 'New info text'

**class** ControllerApp**(**App**):**

**def** build**(**self**):**

**return** Controller**(**info='Hello world'**)**

In this example, a Controller class with 2 properties is created:

* info for receving some text
* label\_wid for receving the label widget

In addition, a do\_action() method is created that will use both of these properties. It will change the change text and info text in the label\_wid widget.

### The layout goes in controller.kv-> Executing this application without a corresponding .kv file will work, but screen will be blank. The Controller class has no widgets in it, it’s just a FloatLayout. The UI can be created around the Controller class in a file named controller.kv, which will be loaded when the ControllerApp is run.

|  |
| --- |
| <Controller>**:**  label\_wid**:** my\_custom\_label  BoxLayout**:**  orientation**:** 'vertical'  padding**:** 20  Button**:**  text**:** 'My controller info is: ' + root.info  on\_press**:** root.do\_action**()**  Label**:**  id**:** my\_custom\_label  text**:** 'My label before button press' |

One label and one button in a vertical BoxLayout. Seems very simple. There are 3 things going on here:

1. Using data from the Controller. As soon as the info property is changed in the controller, the expression text: 'My controller info is: ' + root.info will automatically be re-evaluated which will change the text in the Button.
2. Giving data to the Controller. The expression id: my\_custom\_label is assigning the created Label the id of my\_custom\_label.
3. Creating a custom callback in the Button using the Controller’s on\_press method.
   * root and self are reserved keywords, useable anywhere, root represents the top widget in the rule and self represents the current widget.
   * use any id declared in the rule the same as root and self.
4. on\_press**:** root.do\_action**()**

my\_custom\_label.font\_size = 18

And that’s that. Now when we run main.py, controller.kv will be loaded so that the Button and Label will show up and respond to touch events.

Packaging application on the Android platform

Kivy applications can run on (more or less) any device with OpenGL ES 2.0 (Android 2.2 minimum). This is standard on modern devices; Google reports the requirement is met by more than [99.9% of devices](https://developer.android.com/about/dashboards/index.html). Kivy APKs are normal Android apps that one can distribute like any other, including on stores like the Play store. They behave properly when paused or restarted, may utilise Android services and have access to most of the normal java API.

The Kivy project provides all the necessary tools to package the app on Android, including building ones own standalone APK that may be distributed on a market like the Play store, this is covered fully in the [Create a package for Android](https://kivy.org/doc/stable/guide/packaging-android.html#packaging-android) documentation.

The Android logcat stream provides the normal output of your code (stdout, stderr), as well as the normal Kivy logs, provided by the [Android SDK](http://developer.android.com/sdk/index.html). Enable adb in android device’s developer options, then connect your device to your computer and run:

adb logcat

If the app is packaged already with Buildozer, the adb tool may not be in the $. The line below also runs it:

buildozer android logcat

to run the version installed by Buildozer, or find the SDK tools at $HOME/.buildozer/android/platform. [Kivy Launcher](https://play.google.com/store/apps/details?id=org.kivy.pygame&hl=en) allows to run and debug the application.

CONCLUSION

The Android tools are now quite stable, and should work with practically any device; the minimum requirements are OpenGL ES 2.0 and Android 2.2. As Kivy works fine on most devices and so the list of supported phones/tablets has been retired - all Android devices are likely to work. Kivy framework is powerful and handy, it is easy to learn and use. Crossplatform apps are just one step to any python developer.

REFERENCES

[1] [Kivy org](https://kivy.org/#home)

[2] [Kivy tutorial (Geeks for geeks)](https://www.geeksforgeeks.org/kivy-tutorial/)

# [3] [Andreas Schreiber - Developing Apps for Android and Other Platforms with Kivy and Python](https://www.researchgate.net/profile/Andreas-Schreiber-2)

[4] [Aman Bhoyarkar , Anuja Solanki , Ashutosh Balbudhe - Application Development using Kivy Framework](https://ijarcce.com/papers/application-development-using-kivy-framework/)

[5] [Bc. Ondřej Chrastina - Cross-platform development of smartphone application with the Kivy framework MASTER THESIS](https://is.muni.cz/th/b1nab/dp.pdf)

[6] [Kivy Documentation](https://buildmedia.readthedocs.org/media/pdf/kivy/latest/kivy.pdf)

[7] [Ahmed Fawzy Mohamed Gad - Building Android Apps in Python Using Kivy with Android Studio](https://link.springer.com/content/pdf/bfm%3A978-1-4842-5031-0%2F1.pdf)

# [8] [Mike Driscoll  Build a Mobile Application With the Kivy Python Framework](https://realpython.com/mobile-app-kivy-python/)