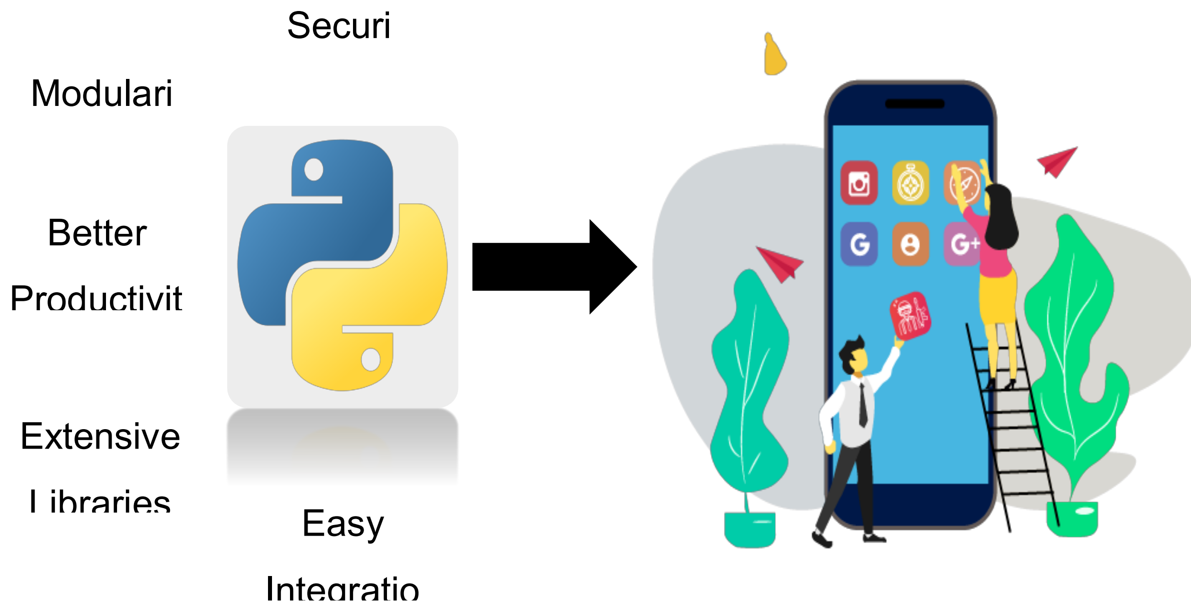
PYTHON FOR MOBILE APPLICATION

Python for Mobile App Development



Python is getting a popular programming language day-by-day. Even some people are using Python for Mobile App Development. Python is highly secure programming language and it also has a lot of libraries already built, so a lot of functions do not require to be coded, and you can simply paste the code from the library and you're done. Moreover, Python code is platform-independent, so the Python code can run on any platform, and that is also one of the main advantages.

Most developers use the Kivy framework to develop a mobile application using Python. With the Kivy framework, you can build cool and intuitive frontends for your mobile application.

Kivy

Kivy is cross-platform

All-Python

Better

Custom User

Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Kivy is a Python-based library and can be used to develop the Frontend for your app. Kivy can be used as a Frontend library and for the backend, one can use Python. So your app is built with Python from A to Z when you use Kivy for the frontend. Now, let's see how the Kivy framework can be useful for our mobile application development.

Kivy is cross-platform.

Kivy framework is cross-platform. So, one code can be used for your Android as well as the iOS application. So, just code once, and you can use the same code in your Android and the iOS application. Kivy reduces coding time and cost.

Custom User Interface Toolkit.

Kivy offers a custom user interface toolkit, which offers various custom text stickers, buttons, textboxes, etc. You can develop a great and eye-pleasing UI for your app with the custom user interface toolkit which allows us access to a lot of basic user-interface elements.

Better consistency.

When you develop your app with the Kivy framework, you get better consistency in terms of app stability and other factors. Python-backed applications are less likely to get crashed or even get attacked by a potential hacker. So, you get both consistency and security when you choose Python for your app development.

All-Python Advantage.

When your app is already backed with Python and you use the Kivy framework for your application's frontend development, you get the "All-Python" advantage. Your backend is already secure with Python, plus you get your frontend developed with the Kivy framework based-on Python. So, your you can say that the app is 'All-Python'.

Basic Kivy code

```
from kivy.app import App
App().run()
```

The most basic Kivy app

```
from kivy.app import App
class WeatherApp(App):
    pass
if __name__ == '__main__’:
    WeatherApp().run()
```

This version uses inheritance to create a new subclass of App called WeatherApp

The starting Python module for all Kivy applications should be named **main.py**, as the build tools you'll use later to automate deployment to mobile devices will look for that

file. Now add a couple of lines of code to this new file, as shown in this slide.

That is it: the most basic Kivy code you could possibly write. It imports an App class, instantiates it, and then calls the run method. **Run** this code by activating your Kivy environment in a terminal and typing python **main.py** (or **kivy main.py** on Mac OS). It will pop up a blank window with a black background.

Second code for creating a new subclass of App called WeatherApp.

KV Language

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Hello World label

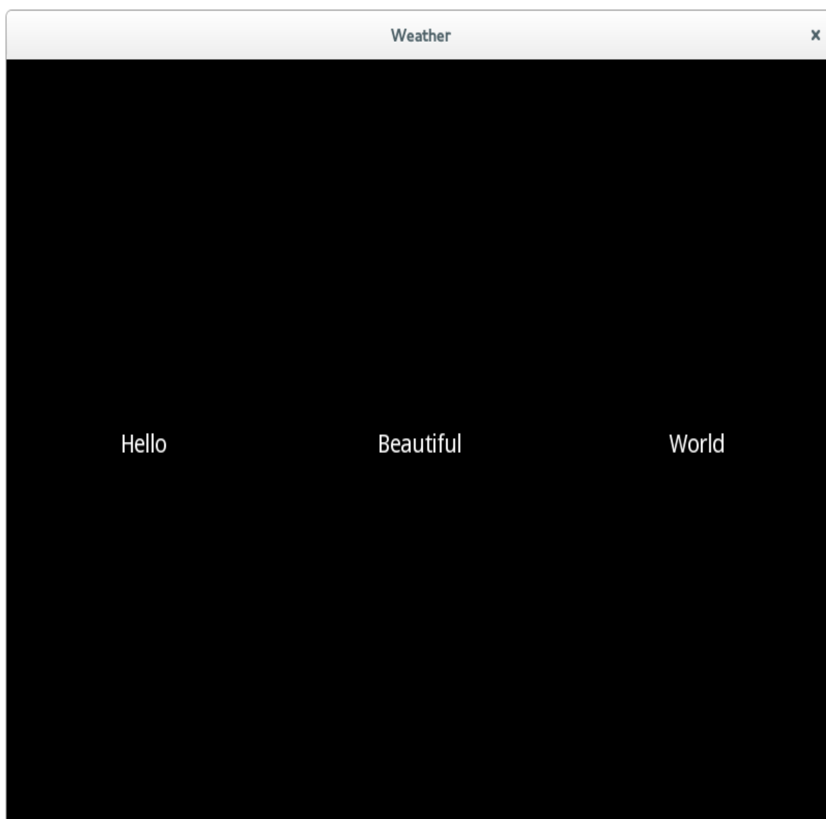
```
Label:
        text: "Hello World"
```

Simple KV language file

weather.kv

This is a very simple KV language file that creates a new Label object and sets its text to the infamous Hello World string. If you now run the python main.py command, you will see the window pop up, still with a black background, but also with the text displayed in its center, as shown in this slide.

Widget



Rendering of basic container widget

```
BoxLayout:
        Label:
                text: "Hello"
        Label:
                text: "Beautiful"
        Label:
                text: "World"
```

Basic container widget

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

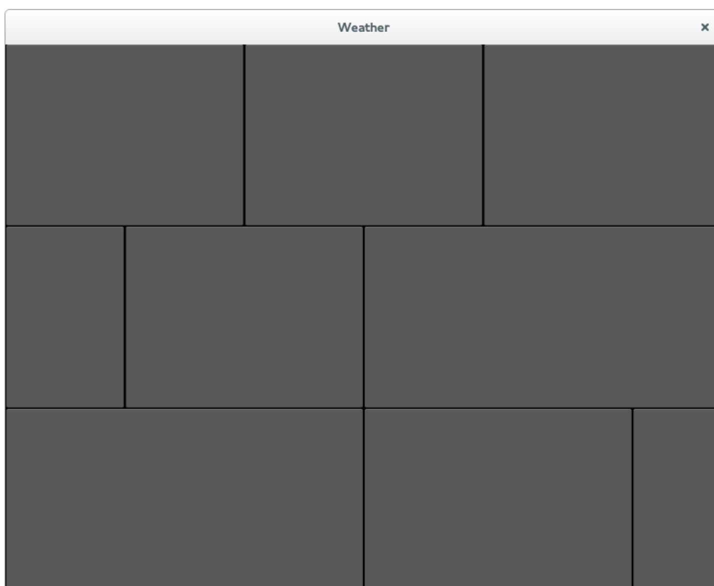
Kivy uses the word **widget** to describe any user interface element. Just a few examples of widgets include:

- The label you rendered in your application;
- The text input field and buttons you'll render shortly;
- Layout classes that comprise other widgets and determine where they should be displayed;
- Complicated tree views such as file pickers;
- Movie and photo renderers;
- Tabbed boxes that display different widgets depending on the selected tab.

The KV language file uses indentation to indicate which **"boxes"** go inside other boxes. The outermost box in a KV language file is called the **root widget**.

The root widget, in this case, is a **BoxLayout** object. **Layout** widgets are essentially containers that know how to hold other widgets and position them in some way. There are three labels supplied, indented, as children of the **BoxLayout**. Each of these **Labels** has an indented block of its own where that widget's properties are configured; in this example, a different value for **text** is supplied for each.

Adjusting Widget Size



Size hints

```
BoxLayout:
        orientation: "vertical"
        BoxLayout:
                Button:
                        size_hint_x: 1
Button:
        size_hint_x: 1
Button:
        size_hint_x: 1
BoxLayout:
Button:
        size_hint_x: 1
Button:
        size_hint_x: 2
Button:
        size_hint_x: 3
BoxLayout:
Button:
        size_hint_x: 1
Button:
        size_hint_x: 0.75
Button:
        size_hint_x: 0.25
```

Size hints

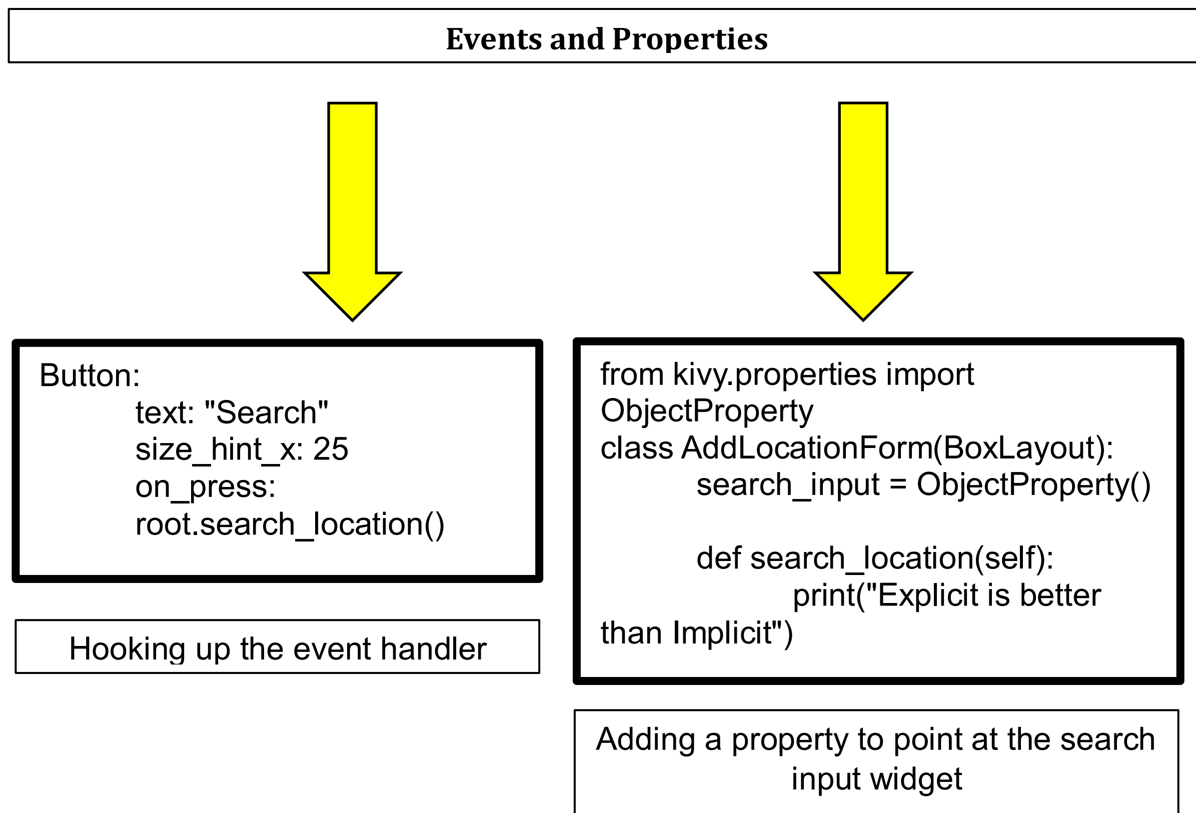
For each type of advice, the child widget can set properties in the **x** dimension (horizontally) and the **y** dimension (vertically). In addition, it is possible to combine the horizontal and vertical settings in the case where you need to explicitly set both of them. This is unecessary with **BoxLayout**, since it always uses maximum space in one direction, but can be useful with other layouts. Thus, there are technically six different proportion advice properties that you can set on any given widget class:

- size_hint_x;
- size_hint_y;

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

• size_hint (tuple of size_hint_x, size_hint_y);
• width;
• height;
• size (tuple of width, height).

The **size_hint** is a proportional measure. If three widgets have the same **size_hint** (and the layout chooses not to ignore that information), they will all be the same size. If one widget's **size_hint** is twice as big as another widget's, then it will be rendered at double the size. The main thing to bear in mind is that the size of a widget is calculated based on the sum of the **size_hint** values for all the widgets. A single widget having a **size_hint** of **1** has no meaning unless you also know that its sibling widget has a **size_hint** of **2** (the first widget will be smaller than the second) or 0.5 (the first widget will be larger).



Dictionary.com defines an event as "something that happens, especially something important." That's a perfect description of events in Kivy. Kivy is firing events all the time, but you only have to pay attention to those that you consider important. Every graphical toolkit has some concept of events. The difference between Kivy and those other toolkits is that in Kivy, event dispatch and handling are sane and uncomplicated.

The **event** handler is accessed as a property on the **Button** object with a prefix of on_. There are specific types of events for different widgets; for a button, the **press** event is kicked off by a mouse press or touch event. When the press event happens, the code following the colon—in this case, **root.search_location()** — is executed as **regular Python code**.

Kivy **properties** are somewhat magical beings. At their most basic, they are special objects that can be attached to widgets in the Python code and have their values accessed and set in the KV

language file. But they add a few special features.

First, Kivy properties have type-checking features. You can always be sure that a String property does not have an integer value, for example. You can also do additional validation, like ensuring that a number is within a specific range.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

More interestingly, Kivy properties can automatically fire events when their values change. This can be incredibly useful, as you will see in later chapters. It's also possible to link the value of one property directly to the value of another property. Thus, when the bound property changes, the linked property's value can be updated to some value calculated from the former.

Finally, Kivy properties contain all sorts of knowledge that is very useful when you're interfacing between the KV language layout file and the actual Python program.

ListView Adapters

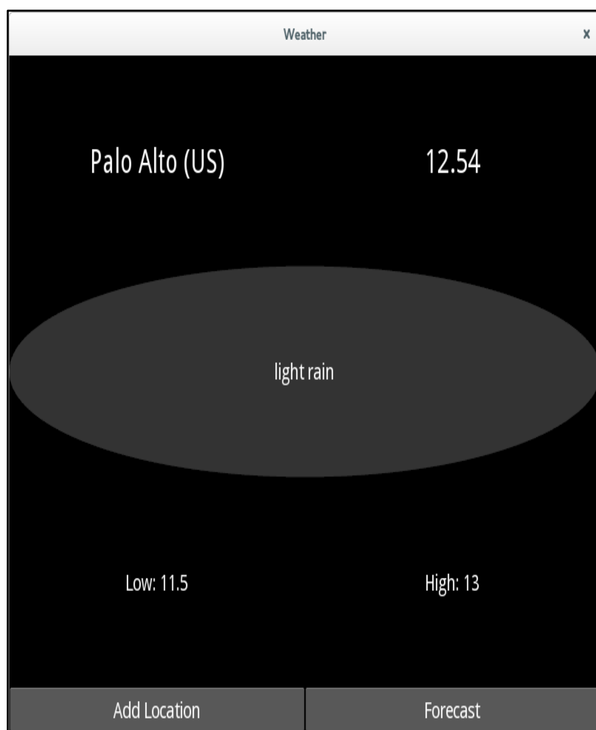
```
#: import ListItemButton kivy.uix.listview.ListItemButton
#: import ListAdapter kivy.adapters.listadapter.ListAdapter
```

Imports for adapter buttons

The Kivy **ListView** API includes full support for managing and displaying selection.

By default, the **ListView** renders a **Label** widget for each string in the list, but **Label** is an inert widget that doesn't care when it gets touched. Luckily, **ListView** can use different classes (including custom classes) as the widget to be displayed for each item. Kivy supplies two classes, **ListItemLabel** and **ListItemButton**, that behave like normal **Label** and **Button** objects but also contain information for tracking selection. Tracking selection is mandatory for **ListView** widgets, so it's almost always a good idea to extend one of these classes, depending on whether you want to just display data (use **ListItemLabel**) or respond to touch events (use **ListItemButton**).

Kivy Graphics



Your first graphics instruction

```
<UnknownConditions@BoxLayout>:
    conditions: ""
    canvas:
        Color:
            rgb: [0.2, 0.2,
    0.2]
        Ellipse:
                pos:
                self.pos
                size:
                self.size
    Label:
        text: root.conditions
```

A simple conditions widget

Kivy provides sophisticated graphics capabilities using OpenGL and SDL instructions. These can be useful if you're creating an interactive game rather than an application with widgets.

There are a few things to notice about this short snippet. Notice the canvas property. If you want to interact with graphics primitives, you need to create instructions on a canvas. Here, you construct

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

two instruction objects, a Color instruction and an Ellipse instruction. These have attributes such as the RGB (red green blue) Color value and the size and position of the Ellipse.

Kivy Graphics

To use Canvas you must have to import:

```
from kivy.graphics import Rectangle, Color
```

Canvas.kv file:

```
<CanvasWidget@Widget>

    #creating canvas
    canvas:
        Color:
            rgba: 0, 0, 1, 1  #Blue

    #size and position of Canvas
    Rectangle:
pos: self.pos
size: self.size
```

Kivy is a platform independent GUI tool in Python. As it can be run on Android, IOS, linux and Windows etc. It is basically used to develop the Android application, but it does not mean that it can not be used on Desktops applications.

The **Canvas** is the root object used for drawing by a Widget. A kivy canvas is not the place where you paint.

Each Widget in Kivy already has a Canvas by default. When you create a widget, you can create all the instructions needed for drawing. If self is your current widget. The instructions Color and Rectangle are automatically added to the canvas object and will be used when the window is drawn.

Animation

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

To use animation you must have to import:

```
from kivy.animation import Animation
```

Simple animation

```
anim = Animation(x=100, y=100)
anim.start(widget)
```

Multiple properties and transitions

```
anim = Animation(x=50, size=(80, 80), t='in_quad')
anim.start(widget)
```

Sequential animation

```
anim = Animation(x=50) + Animation(size=(80, 80), duration=2.)
anim.start(widget)
```

Repeating animation

```
anim = Animation(...) + Animation(...)
anim.repeat = True
anim.start(widget)
```

Parallel animation

```
anim = Animation(pos=(80, 10))
anim &= Animation(size=(800, 800), duration=2.)
anim.start(widget)
```

Animation and **AnimationTransition** are used to animate **Widget** properties. You must specify at least a property name and target value. To use an Animation, follow these steps:

- Setup an Animation object;
- Use the Animation object on a Widget.

Simple animation.

To animate a Widget's x or y position, simply specify the target x/y values where you want the widget positioned at the end of the animation.

The animation will last for 1 second unless **duration** is specified. When **anim.start()** is called, the Widget will move smoothly from the current x/y position to (100, 100).

Multiple properties and transitions.

You can animate multiple properties and use built-in or custom transition functions using transition (or the t= shortcut). For example, to animate the position and size using the 'in_quad' transition.

Note that the **t=** parameter can be the string name of a method in the **AnimationTransition** class or your own animation function.

Sequential animation.

To join animations sequentially, use the '+' operator. The following example will animate to x=50 over 1 second, then animate the size to (80, 80) over the next two seconds.

Parallel animation.

To join animations in parallel, use the **'&'** operator. The following example will animate the position to (80, 10) over 1 second, whilst in parallel animating the size to (800, 800).

Keep in mind that creating overlapping animations on the same property may have unexpected results. If you want to apply multiple animations to the same property, you should either schedule them sequentially (via the **'+'** operator or using the **on_complete callback**) or cancel previous animations using the **cancel_all** method.

Repeating animation.

To set an animation to repeat, simply set the **Sequence.repeat** property to **True**.

For flow control of animations such as stopping and cancelling, use the methods already in place in the animation module.

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Releasing to Android and iOS

Android		iOS
1. Install buildozer; 2. Create the spec file; 3. Generate the APK, use the APK file in ./bin/ folder to install; 4. Or plug your phone, generate and install in one go; 5. Deploy on production environment will need to add a sign tuso the package.	**BUILDOZER**	1. Install Xcode and related SDK; 2. Install libs for building; 3. Install Cython; 4. Download kivy-ios and install; 5. Create the Xcode project: (your entry file must be named as main.py) 6. You will see the a folder named <title>-ios, open it and open the project via xxxxxx.xcodeproj 7. Go to apple developer center and register as a developer 8. Open your project settings 9. Run the project

Kivy has a custom-built deployment tool called Buildozer.

Buildozer is a tool that aim to package mobiles application easily. It automates the entire build process, download the prerequisites like python-for-android, Android SDK, NDK, etc.

Buildozer manage a file named **buildozer.spec** in your application directory, describing your application requirements and settings such as title, icon, included modules etc. It will use the specification file to create a package for Android, iOS, and more.

Currently, Buildozer supports packaging for:

- Android: via Python for Android;
- iOS: via Kivy iOS;
- Supporting others platform is in the roadmap.

REFERENCES

1. Link: [https://yourstory.com/mystory/choose-python-mobile-app-development]
2. Creating Apps in Kivy (Mobile with Python) by Dusty Phillips
3. Link: [https://www.geeksforgeeks.org/python-canvas-in-kivy-using-kv-file/]
4. Link: [https://www.albertgao.xyz/2017/06/14/how-to-deploy-kivy-app-to-ios-and-android/]

This document has been produced with the support of the EUROPEAN COMMISSION under the ERASMUS+ Programme, KA2 – Capacity Building in the Field of Higher Education: 598092-EPP-1-2018-1-BG-EPPKA2-CBHE-SP.
It reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.