

Real Python

Build a Mobile Application With the Kivy Python Framework

by Mike Driscoll | Nov 04, 2019 | 49 Comments | front-end | gui | intermediate

Mark as Completed



Share Share Email

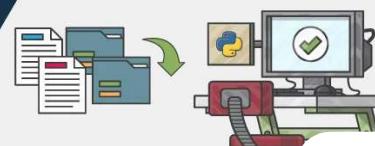
Table of Contents

- [Understanding the Kivy Framework](#)
- [Installing Kivy](#)
- [Working With Kivy Widgets](#)
 - [Running a “Hello, Kivy!” Program](#)
 - [Displaying an Image](#)
- [Laying Out the UI](#)
- [Adding Events](#)
- [Using the KV Language](#)
- [Creating a Kivy Application](#)
- [Packaging Your App for Android](#)
- [Packaging Your App for iOS](#)
- [Packaging Your App for Windows](#)
- [Packaging Your App for macOS](#)
- [Conclusion](#)
- [Further Reading](#)

Free PDF Download: Python 3 Cheat Sheet

[Download Now](#)

realpython.com



Remove ads

Help

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Build Cross-Platform GUI Apps With Kivy](#)

These days, developers are highly likely to be working on a mobile or web application. Python doesn't have built-in mobile development capabilities, but there are packages you can use to create mobile applications, like Kivy, [PyQt](#), or even Beeware's [Toga](#) library.

These libraries are all major players in the Python mobile space. However, there are some benefits you'll see if you choose to create mobile applications with **Kivy**. Not only will your application look the same on all platforms, but you also won't need to compile your code after every change. What's more, you'll be able to use Python's clear syntax to build your applications.

In this tutorial, you'll learn how to:

- Work with Kivy widgets
- Lay out the UI
- Add events
- Use the KV language
- Create a calculator application
- Package your application for iOS, Android, Windows, and macOS

This tutorial assumes you're familiar with object-oriented programming. If you're not, then check out [Object-Oriented Programming \(OOP\) in Python 3](#).

Let's get started!

Free Download: [Get a sample chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Understanding the Kivy Framework

Kivy was first released in early 2011. This cross-platform Python framework can be deployed to Windows, Mac, Linux, and Raspberry Pi. It supports [multitouch](#) events in addition to regular keyboard and mouse inputs. Kivy even supports GPU acceleration of its graphics, since they're built using OpenGL ES2. The project uses the MIT license, so you can use this library for free and commercial software.

When you create an application with Kivy, you're creating a [Natural User Interface](#) or **NUI**. The idea behind a Natural User Interface is that the user can easily learn how to use your software with little to no instruction.

Kivy does not attempt to use native controls or widgets. All of its widgets are custom-drawn. This means that Kivy applications will look the same across all platforms. However, it also means that your app's look and feel will differ from your user's native applications. This could be a benefit or a drawback, depending on your audience.

Improve Your Python with Python Tricks

realpython.com



[i Remove ads](#)

Installing Kivy

Kivy has many dependencies, so it's recommended that you install it into a Python virtual environment. You can use either Python's built-in [venv](#) library or the [virtualenv](#) package. If you've never used a Python virtual environment before, then check out [Python Virtual Environments: A Primer](#).

Here's how you can create a Python virtual environment:

Shell



```
$ python3 -m venv my_kivy_project
```

This will copy your Python 3 executable into a folder called `my_kivy_project` and add a few other subfolders to that directory.

To use your virtual environment, you need to activate it. On Mac and Linux, you can do that by executing the following while inside the `my_kivy_project` folder:

Shell

```
$ source bin/activate
```



The command for Windows is similar, but the location of the activate script is inside of the `Scripts` folder instead of `bin`.

Now that you have an activated Python virtual environment, you can run `pip` to install Kivy. On Linux and Mac, you'll run the following command:

Shell

```
$ python -m pip install kivy
```



On Windows, installation is a bit more complex. Check out the official documentation for how to install [Kivy on Windows](#). (Mac users can also download a `dmg` file and install Kivy that way.)

If you run into any issues installing Kivy on your platform, then see the Kivy download page for additional [instructions](#).

Working With Kivy Widgets

A **widget** is an onscreen control that the user will interact with. All graphical user interface toolkits come with a set of widgets. Some common widgets that you may have used include buttons, combo boxes, and tabs. Kivy has many widgets built into its framework.

Running a “Hello, Kivy!” Program

To see how Kivy works, take a look at the following “Hello, World!” application:

Python

```
from kivy.app import App
from kivy.uix.label import Label

class MainApp(App):
    def build(self):
        label = Label(text='Hello from Kivy',
                      size_hint=(.5, .5),
                      pos_hint={'center_x': .5, 'center_y': .5})

        return label

if __name__ == '__main__':
    app = MainApp()
    app.run()
```

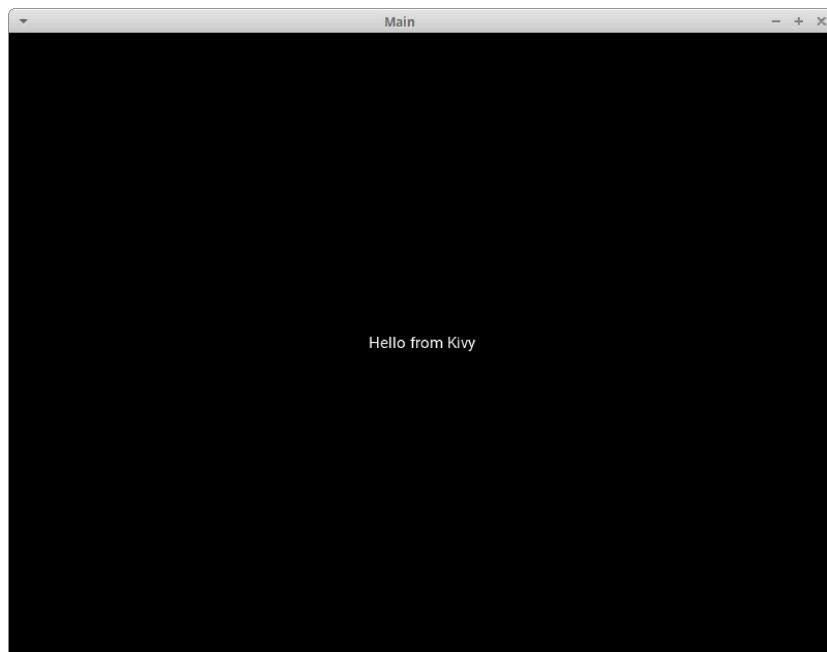
Every Kivy application needs to subclass `App` and override `build()`. This is where you'll put your UI code or make calls to other functions that define your UI code. In this case, you create a `Label` widget and pass in its `text`, `size_hint`, and `pos_hint`. These last two arguments are not required.

`size_hint` tells Kivy the proportions to use when creating the widget. It takes two numbers:

1. The first number is the x size hint and refers to the **width** of the control.
2. The second number is the y size hint and refers to the **height** of the control.

Both of these numbers can be anywhere between 0 and 1. The default value for both hints is 1. You can also use `pos_hint` to position the widget. In the code block above, you tell Kivy to center the widget on the x and y axes.

To make the application run, you instantiate your `MainApp` class and then call `run()`. When you do so, you should see the following on your screen:



Kivy also outputs a lot of text to `stdout`:

Shell

```
[INFO    ] [Logger      ] Record log in /home/mdriscoll/.kivy/logs/kivy_19-06-07_2.txt
[INFO    ] [Kivy        ] v1.11.0
[INFO    ] [Kivy        ] Installed at "/home/mdriscoll/code/test/lib/python3.6/site-packages/kivy/__init__.py"
[INFO    ] [Python       ] v3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0]
[INFO    ] [Python       ] Interpreter at "/home/mdriscoll/code/test/bin/python"
[INFO    ] [Factory     ] 184 symbols loaded
[INFO    ] [Image        ] Providers: img_tex, img_dds, img_sdl2, img_gif (img_pil, img_ffpyplayer ignored)
[INFO    ] [Text         ] Provider: sdl2(['text_pango'] ignored)
[INFO    ] [Window       ] Provider: sdl2(['window_egl_rpi'] ignored)
[INFO    ] [GL           ] Using the "OpenGL" graphics system
[INFO    ] [GL           ] Backend used <sdl2>
[INFO    ] [GL           ] OpenGL version <b'4.6.0 NVIDIA 390.116'>
[INFO    ] [GL           ] OpenGL vendor <b'NVIDIA Corporation'>
[INFO    ] [GL           ] OpenGL renderer <b'NVS 310/PCIe/SSE2'>
[INFO    ] [GL           ] OpenGL parsed version: 4, 6
[INFO    ] [GL           ] Shading version <b'4.60 NVIDIA'>
[INFO    ] [GL           ] Texture max size <16384>
[INFO    ] [GL           ] Texture max units <32>
[INFO    ] [Window       ] auto add sdl2 input provider
[INFO    ] [Window       ] virtual keyboard not allowed, single mode, not docked
[INFO    ] [Base          ] Start application main loop
[INFO    ] [GL           ] NPOT texture support is available
```

This is useful for debugging your application.

Next, you'll try adding an `Image` widget and see how that differs from a `Label`.

Python Dependency Management Pitfalls

A free email class

realpython.com



[Remove ads](#)

Displaying an Image

Kivy has a couple of different image-related widgets to choose from. You can use `Image` to load local images from your hard drive or `AsyncImage` to load an image from a URL. For this example, you'll stick with the standard `Image` class:

Python

```
from kivy.app import App
from kivy.uix.image import Image

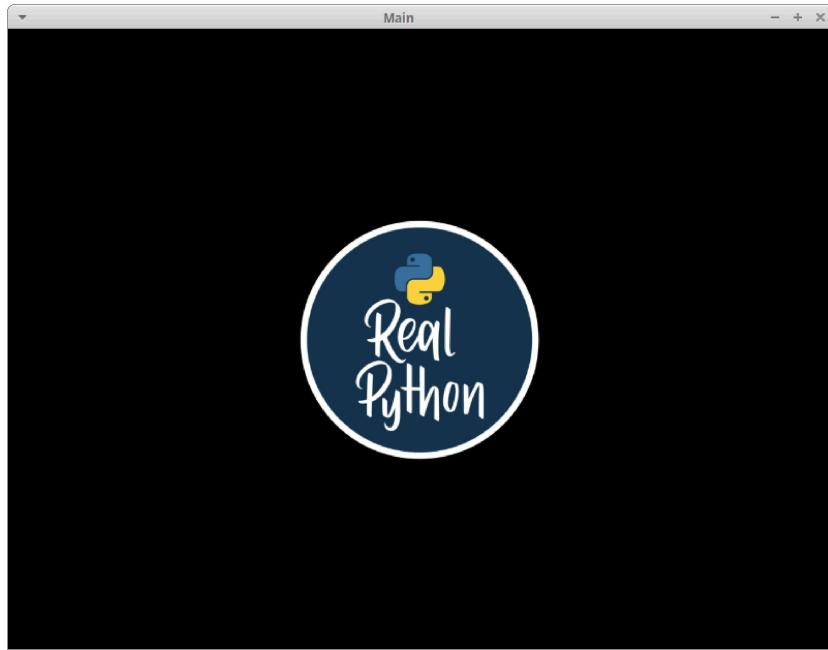
class MainApp(App):
    def build(self):
        img = Image(source='/path/to/real_python.png',
                    size_hint=(1, .5),
                    pos_hint={'center_x':.5, 'center_y':.5})

    return img

if __name__ == '__main__':
    app = MainApp()
    app.run()
```

In this code, you import `Image` from the `kivy.uix.image` sub-package. The `Image` class takes a lot of different parameters, but the one that you want to use is `source`. This tells Kivy which image to load. Here, you pass a fully-qualified path to the image. The rest of the code is the same as what you saw in the previous example.

When you run this code, you'll see something like the following:



The text from the previous example has been replaced with an image.

Now you'll learn how to add and arrange multiple widgets in your application.

Laying Out the UI

Each GUI framework that you use has its own method of arranging widgets. For example, in [wxPython](#) you'll use sizers, while in [Tkinter](#) you use a layout or geometry manager. With Kivy, you'll use **Layouts**. There are several different types of Layouts that you can use. Here are some of the most common ones:

- `BoxLayout`
- `FloatLayout`
- `GridLayout`

You can search Kivy's [documentation](#) for a full list of available Layouts. You can also look in `kivy.uix` for the actual source code.

Try out the BoxLayout with this code:

Python

```
import kivy
import random

from kivy.app import App
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout

red = [1,0,0,1]
green = [0,1,0,1]
blue = [0,0,1,1]
purple = [1,0,1,1]

class HBoxLayoutExample(App):
    def build(self):
        layout = BoxLayout(padding=10)
        colors = [red, green, blue, purple]

        for i in range(5):
            btn = Button(text="Button #%" % (i+1),
                         background_color=random.choice(colors)
                         )

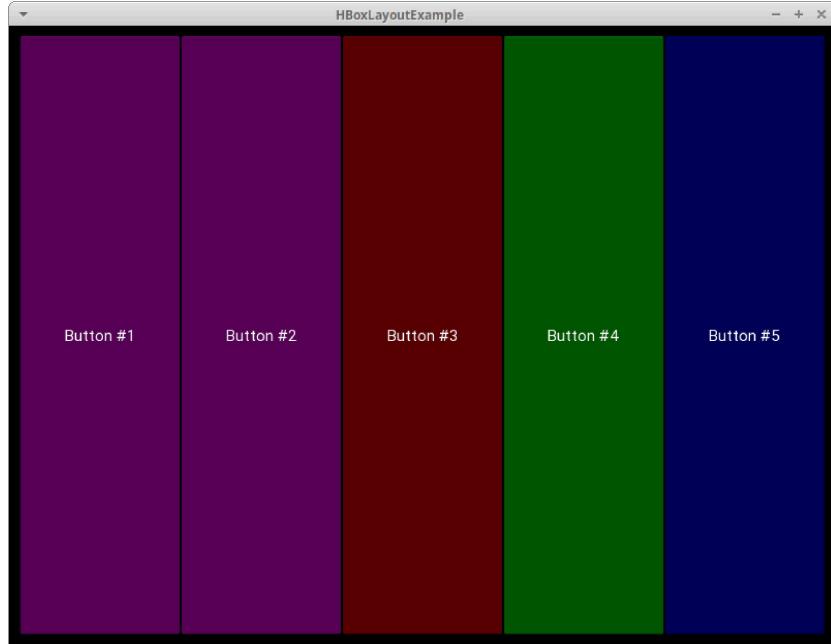
            layout.add_widget(btn)

        return layout

if __name__ == "__main__":
    app = HBoxLayoutExample()
    app.run()
```

Here, you import BoxLayout from kivy.uix.boxlayout and instantiate it. Then you create a list of colors, which are themselves lists of Red-Blue-Green (RGB) colors. Finally, you loop over a `range` of 5, creating a button `btn` for each iteration. To make things a bit more fun, you set the `background_color` of the button to a random color. You then add the button to your layout with `layout.add_widget(btn)`.

When you run this code, you'll see something like this:



There are 5 randomly-colored buttons, one for each iteration of your `for` loop.

When you create a layout, there are a few arguments you should know:

- **padding:** You can specify the padding in pixels between the layout and its children in one of three ways:

1. **A four-argument list:** [padding_left, padding_top, padding_right, padding_bottom]

2. **A two-argument list:** [padding_horizontal, padding_vertical]

3. **A singular argument:** padding=10

- **spacing:** You can add space between the children widgets with this argument.
- **orientation:** You can change the default orientation of the BoxLayout from horizontal to vertical.

5 Thoughts on Mastering Python

A free email class for Python developers

realpython.com



[Remove ads](#)

Adding Events

Like most GUI toolkits, Kivy is mostly **event-based**. The framework responds to user keypresses, mouse events, and touch events. Kivy has the concept of a **Clock** that you can use to schedule function calls for some time in the future.

Kivy also has the concept of [Properties](#), which works with the [EventDispatcher](#). Properties help you do validation checking. They also let you fire events whenever a widget changes its size or position.

Let's add a **button event** to your button code from earlier:

Python

```
from kivy.app import App
from kivy.uix.button import Button

class MainApp(App):
    def build(self):
        button = Button(text='Hello from Kivy',
                        size_hint=(.5, .5),
                        pos_hint={'center_x': .5, 'center_y': .5})
        button.bind(on_press=self.on_press_button)

        return button

    def on_press_button(self, instance):
        print('You pressed the button!')

if __name__ == '__main__':
    app = MainApp()
    app.run()
```

In this code, you call `button.bind()` and link the `on_press` event to `MainApp.on_press_button()`. This method implicitly takes in the widget `instance`, which is the `button` object itself. Finally, a message will print to `stdout` whenever the user presses your button.

Using the KV Language

Kivy also provides a design language called **KV** that you can use with your Kivy applications. The KV language lets you separate your interface design from the application's logic. This follows the [separation of concerns](#) principle and is part of the [Model-View-Controller](#) architectural pattern. You can update the previous example to use the KV language:

Python

```

from kivy.app import App
from kivy.uix.button import Button

class ButtonApp(App):
    def build(self):
        return Button()

    def on_press_button(self):
        print('You pressed the button!')

if __name__ == '__main__':
    app = ButtonApp()
    app.run()

```

This code might look a bit odd at first glance, as it creates a `Button` without setting any of its attributes or binding it to any events. What's happening here is that Kivy will automatically look for a file that has the same name as the class in lowercase, without the `App` part of the class name.

In this case, the class name is `ButtonApp`, so Kivy will look for a file named `button.kv`. If that file exists and is properly formatted, then Kivy will use it to load up the UI. Go ahead and create this file and add the following code:

Text

```

1 <Button>:
2     text: 'Press me'
3     size_hint: (.5, .5)
4     pos_hint: {'center_x': .5, 'center_y': .5}
5     on_press: app.on_press_button()

```

Here's what each line does:

- **Line 1** matches the `Button` call in your Python code. It tells Kivy to look into the instantiated object for a button definition.
- **Line 2** sets the button's text.
- **Line 3** sets the width and height with `size_hint`.
- **Line 4** sets the button's position with `pos_hint`.
- **Line 5** sets the `on_press` event handler. To tell Kivy where the event handler is, you use `app.on_press_button()`. Here, Kivy knows will look in the `Application` class for a method called `.on_press_button()`.

You can set up all of your widgets and layouts inside one or more KV language files. The KV language also supports importing Python modules in KV, creating dynamic classes, and much more. For full details, check out Kivy's [guide to the KV Language](#).

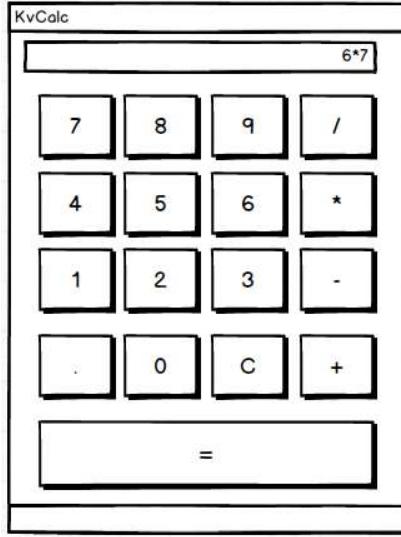
Now you're ready to create a real application!

Creating a Kivy Application

One of the best ways to learn a new skill is by creating something useful. With that in mind, you'll use Kivy to build a calculator that supports the following operations:

- Addition
- Subtraction
- Multiplication
- Division

For this application, you'll need a series of buttons in some kind of layout. You'll also need a box along the top of your app to display the equations and their results. Here's a sketch of your calculator:



Now that you have a goal for the UI, you can go ahead and write the code:

Python

```

1  from kivy.app import App
2  from kivy.uix.boxlayout import BoxLayout
3  from kivy.uix.button import Button
4  from kivy.uix.textinput import TextInput
5
6  class MainApp(App):
7      def build(self):
8          self.operators = ["/", "*", "+", "-"]
9          self.last_was_operator = None
10         self.last_button = None
11         main_layout = BoxLayout(orientation="vertical")
12         self.solution = TextInput(
13             multiline=False, readonly=True, halign="right", font_size=55
14         )
15         main_layout.add_widget(self.solution)
16         buttons = [
17             ["7", "8", "9", "/"],
18             ["4", "5", "6", "*"],
19             ["1", "2", "3", "-"],
20             [".", "0", "C", "+"],
21         ]
22         for row in buttons:
23             h_layout = BoxLayout()
24             for label in row:
25                 button = Button(
26                     text=label,
27                     pos_hint={"center_x": 0.5, "center_y": 0.5},
28                 )
29                 button.bind(on_press=self.on_button_press)
30                 h_layout.add_widget(button)
31             main_layout.add_widget(h_layout)
32
33         equals_button = Button(
34             text "=", pos_hint={"center_x": 0.5, "center_y": 0.5}
35         )
36         equals_button.bind(on_press=self.on_solution)
37         main_layout.add_widget>equals_button
38
39     return main_layout

```

Here's how your calculator code works:

- In **lines 8 to 10**, you create a list of operators and a couple of handy values, `last_was_operator` and `last_button`, that you'll use later on.
- In **lines 11 to 15**, you create a top-level layout `main_layout` and add a read-only `TextInput` widget to it.

- In **lines 16 to 21**, you create a nested list of lists containing most of your buttons for the calculator.
- In **line 22**, you start a `for` loop over those buttons. For each nested list you'll do the following:
 - In **line 23**, you create a `BoxLayout` with a horizontal orientation.
 - In **line 24**, you start another `for` loop over the items in the nested list.
 - In **lines 25 to 39**, you create the buttons for the row, bind them to an event handler, and add the buttons to the horizontal `BoxLayout` from line 23.
 - In **line 31**, you add this layout to `main_layout`.
- In **lines 33 to 37**, you create the equals button (=), bind it to an event handler, and add it to `main_layout`.

The next step is to create the `.on_button_press()` event handler. Here's what that code looks like:

Python

```

41 def on_button_press(self, instance):
42     current = self.solution.text
43     button_text = instance.text
44
45     if button_text == "C":
46         # Clear the solution widget
47         self.solution.text = ""
48     else:
49         if current and (
50             self.last_was_operator and button_text in self.operators):
51             # Don't add two operators right after each other
52             return
53         elif current == "" and button_text in self.operators:
54             # First character cannot be an operator
55             return
56         else:
57             new_text = current + button_text
58             self.solution.text = new_text
59             self.last_button = button_text
60             self.last_was_operator = self.last_button in self.operators

```

Most of the widgets in your application will call `.on_button_press()`. Here's how it works:

- **Line 41** takes the `instance` argument so you can access which widget called the function.
- **Lines 42 and 43** extract and store the value of the `solution` and the `button text`.
- **Lines 45 to 47** check to see which button was pressed. If the user pressed C, then you'll clear the `solution`. Otherwise, move on to the `else` statement.
- **Line 49** checks if the `solution` has any pre-existing value.
- **Line 50 to 52** check if the last button pressed was an operator button. If it was, then `solution` won't be updated. This is to prevent the user from having two operators in a row. For example, 1 */ is not a valid statement.
- **Lines 53 to 55** check to see if the first character is an operator. If it is, then `solution` won't be updated, since the first value can't be an operator value.
- **Lines 56 to 58** drop to the `else` clause. If none of the previous conditions are met, then update `solution`.
- **Line 59** sets `last_button` to the label of the last button pressed.
- **Line 60** sets `last_was_operator` to `True` or `False` depending on whether or not it was an operator character.

The last bit of code to write is `.on_solution()`:

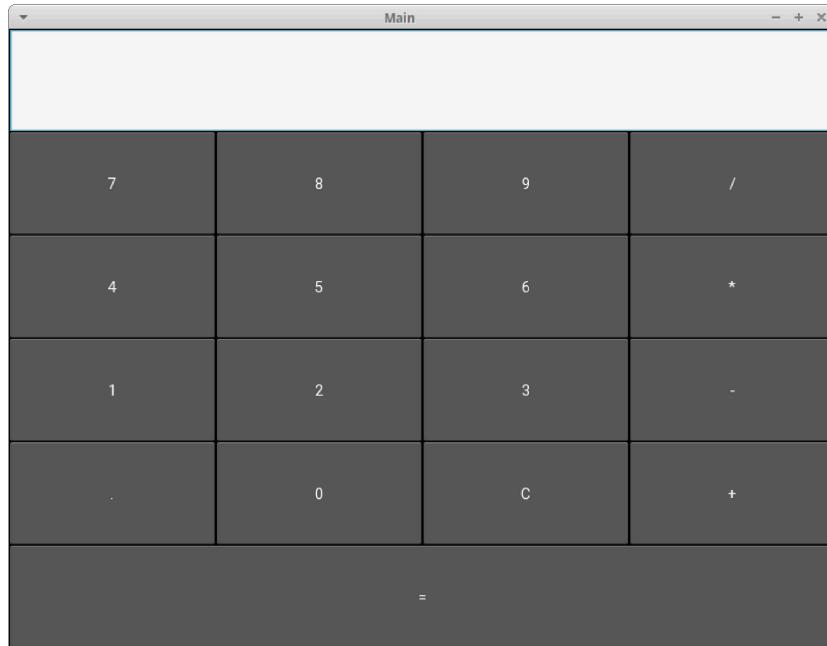
Python

```
62 | def on_solution(self, instance):
63 |     text = self.solution.text
64 |     if text:
65 |         solution = str(eval(self.solution.text))
66 |         self.solution.text = solution
```

Once again, you grab the current text from `solution` and use [Python's built-in `eval\(\)`](#) to execute it. If the user created a formula like `1+2`, then `eval()` will run your code and return the result. Finally, you set the result as the new value for the `solution` widget.

Note: `eval()` is somewhat [dangerous](#) because it can run arbitrary code. Most developers avoid using it because of that fact. However, since you're only allowing integers, operators, and the period as input to `eval()`, it's safe to use in this context.

When you run this code, your application will look like this on a desktop computer:



To see the full code for this example, expand the code block below.

[Complete Code Example](#)

[Show/Hide](#)

It's time to deploy your application!



[Online Python Training for Teams »](#)

[i Remove ads](#)

Packaging Your App for Android

Now that you've finished the code for your application, you can share it with others. One great way to do that is to turn your code into an application that can run on your Android phone. To accomplish this, first you'll need to install a package called `buildozer` with `pip`:

Shell

```
$ pip install buildozer
```

Then, create a new folder and navigate to it in your terminal. Once you're there, you'll need to run the following command:

Shell



```
$ buildozer init
```

This will create a `buildozer.spec` file that you'll use to configure your build. For this example, you can edit the first few lines of the spec file as follows:

Text

```
[app]

# (str) Title of your application
title = KvCalc

# (str) Package name
package.name = kvcalc

# (str) Package domain (needed for android/ios packaging)
package.domain = org.kvcalc
```

Feel free to browse the rest of the file to see what else you can change.

At this point, you're almost ready to build your application, but first, you'll want to install the [dependencies](#) for `buildozer`. Once those are installed, copy your calculator application into your new folder and rename it to `main.py`. This is required by `buildozer`. If you don't have the file named correctly, then the build will fail.

Now you can run the following command:

Shell

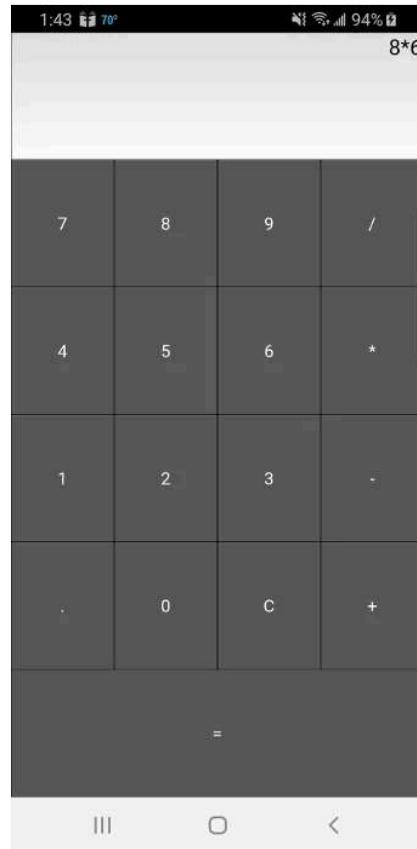


```
$ buildozer -v android debug
```

The build step takes a long time! On my machine, it took 15 to 20 minutes. Depending on your hardware, it may take even longer, so feel free to grab a cup of coffee or go for a run while you wait. `Buildozer` will download whatever Android SDK pieces it needs during the build process. If everything goes according to plan, then you'll have a file named something like `kvcalc-0.1-debug.apk` in your `bin` folder.

The next step is to connect your Android phone to your computer and copy the `apk` file to it. Then you can open the file browser on your phone and click on the `apk` file. Android should ask you if you'd like to install the application. You may see a warning since the app was downloaded from outside Google Play, but you should still be able to install it.

Here's the calculator running on my Samsung S9:



The buildozer tool has several other commands you can use. Check out the [documentation](#) to see what else you can do.

You can also package the app using `python-for-android` if you need more fine-grained control. You won't cover this here, but if you're interested, check out the project's [quickstart](#).

Packaging Your App for iOS

The instructions for building an application for iOS are a bit more complex than Android. For the most up-to-date information, you should always use Kivy's official packaging [documentation](#). You'll need to run the following commands before you can package your application for iOS on your Mac:

Shell

```
$ brew install autoconf automake libtool pkg-config  
$ brew link libtool  
$ sudo easy_install pip  
$ sudo pip install Cython==0.29.10
```

Once those are all installed successfully, you'll need to compile the distribution using the following commands:

Shell

```
$ git clone git://github.com/kivy/kivy-ios  
$ cd kivy-ios  
$ ./toolchain.py build python3 kivy
```

If you get an error that says `iphonesimulator` can't be found, then see this [StackOverflow answer](#) for ways to solve that issue. Then try running the above commands again.

If you run into SSL errors, then you probably don't have Python's OpenSSL setup. This command should fix that:

Shell

```
$ cd /Applications/Python\ 3.7/  
$ ./Install\ Certificates.command
```

Now go back and try running the `toolchain` command again.

Once you've run all the previous commands successfully, you can create your Xcode project using the `toolchain` script. Your main application's entry point *must* be named `main.py` before you create the Xcode project. Here is the command you'll run:

Shell

```
./toolchain.py create <title> <app_directory>
```

There should be a directory named `title` with your Xcode project in it. Now you can open that project in Xcode and work on it from there. Note that if you want to submit your application to the App Store, then you'll have to create a developer account at developer.apple.com and pay their yearly fee.



The screenshot shows a dark-themed web page with a blue header bar. On the left, there is a logo for "Real Python" featuring a yellow Python icon. To the right of the logo, the text "Real Python for Teams »" is displayed. Below the header, there is a small "Remove ads" button with a question mark icon. The rest of the page is mostly blank white space.

Packaging Your App for Windows

You can package your Kivy application for Windows using PyInstaller. If you've never used it before, then check out [Using PyInstaller to Easily Distribute Python Applications](#).

You can install PyInstaller using pip:

Shell

```
$ pip install pyinstaller
```

The following command will package your application:

Shell

```
$ pyinstaller main.py -w
```

This command will create a Windows executable and several other files. The `-w` argument tells PyInstaller that this is a windowed application, rather than a command-line application. If you'd rather have PyInstaller create a single executable file, then you can pass in the `--onefile` argument in addition to `-w`.

Packaging Your App for macOS

You can use [PyInstaller](#) to create a Mac executable just like you did for Windows. The only requirement is that you run this command on a Mac:

Shell

```
$ pyinstaller main.py -w --onefile
```

This will create a single file executable in the `dist` folder. The executable will be the same name as the Python file that you passed to PyInstaller. If you'd like to reduce the file size of the executable, or you're using GStreamer in your application, then check out Kivy's [packaging page for macOS](#) for more information.

Conclusion

Kivy is a really interesting GUI framework that you can use to create desktop user interfaces and mobile applications on both iOS and Android. Kivy applications will not look like the native apps on any platform. This can be an advantage if you want your application to look and feel different from the competition!

In this tutorial, you learned the basics of Kivy including how to add widgets, hook up events, lay out multiple widgets, and use the KV language. Then you created your first Kivy application and learned how to distribute it on other platforms, including mobile!

There are many widgets and concepts about Kivy that you didn't cover here, so be sure to check out Kivy's [website](#) for tutorials, sample applications, and much more.

Further Reading

To learn more about Kivy, check out these resources:

- [Kivy Programming Guide](#)
- [Kivy Packaging Documentation](#)
- [Build Desktop GUI Apps Using Python](#)

To see how you might create a desktop application with another Python GUI framework, check out [How to Build a Python GUI Application With wxPython](#).

Mark as Completed   

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Build Cross-Platform GUI Apps With Kivy](#)



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

kamaljp@gmail.com

[Send Me Python Tricks »](#)

About Mike Driscoll