Kivy – Interactive Applications and Games in Python
Second Edition

Create responsive cross-platform UI/UX applications and games in Python using the open source Kivy library

Roberto Ulloa
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'GUI Basics – Building an Interface'
- A synopsis of the book’s content
- More information on Kivy – Interactive Applications and Games in Python Second Edition
About the Author

Roberto Ulloa has a diverse academic record in multiple disciplines within the field of computer science. Currently, he is working with artificial societies as part of his PhD thesis at the University of Western Ontario. He obtained an MSc degree from the University of Costa Rica and taught programming and computer networking there. He has earned a living as a web developer, working with Python/Django and PHP/Wordpress. He collaborates with various researchers while also working on his own projects, including his blog (http://robertour.com). He constantly worries that the Internet has already become aware of itself and that we are not able to communicate with it because of the improbability of it being able to speak any of the 6,000-plus odd human languages that exist on the planet.
Preface

Mobile devices have transformed the way applications are perceived. They have increased in interaction types; the user now expects gestures, multi-touches, animations, responsiveness, virtual keyboards, and magic-pens. Moreover, compatibility has become a must if you want to avoid the barriers imposed by major operating systems. Kivy is an open source Python solution that covers these market needs with an easy-to-learn and rapid development approach. Kivy continues to grow fast and two versions have been released since the first publication of this book in September 2013. Thanks to an enthusiastic community, Kivy is making its way in an extremely competitive territory in which it stands out for offering both a cross-platform and efficient alternative to native development and HTML5.

This book introduces you to the Kivy world, covering a large variety of important topics related to interactive applications and games development. The components presented in this book were selected according to their usefulness for developing state-of-art applications and also for serving as an example of broader Kivy functionalities. Following this approach, the book covers a big part of the Kivy library.

This book provides you with examples to understand their use and how to integrate the three projects that come with this book. The first one, the comic creator, exemplifies how to build a user interface (Chapter 1, GUI Basics – Building an Interface), how to draw vector shapes in the screen (Chapter 2, Graphics – the Canvas), how to bind user interactions with pieces codes (Chapter 3, Widget Events – Binding Actions), and other components related to improving the user experience (Chapter 4, Improving the User Experience). The second project, Invaders Revenge, is an interactive game that introduces you to the use of animations, scheduling of tasks, keyboard events, and multi-touch control (Chapter 5, Invaders Revenge – an Interactive Multi-touch Game). The third project, Kivy Player, teaches how we can control video streams with a modern design and responsive interactions to maximize the use of the screen (Chapter 6, Kivy Player – a TED Video Streamer).
Occasionally, this book explains some technical but important Kivy concepts that are related to the Kivy class structure and implementation, or the order and strategies to draw on the screen. These explanations give the reader some insights into the Kivy internals that will help them solve potential problems when they develop their own projects. Even though they are not necessary for the comprehension of the main topics of this book, they will become important lessons when the reader faces new situations implementing their own applications.

This book grabs the reader's attention by stating interesting programming scenarios. The sections are generally short and straightforward, making the learning process constant. These short sections will also serve as a reference when the reader finishes the book. However, serving as a reference doesn't prevent the text from achieving the main goal, which is teaching bigger projects that connect the small topics. At the end of this book, the reader will feel comfortable to start their own project.

**What this book covers**

*Chapter 1, GUI Basics – Building an Interface,* introduces the basic components and layouts of Kivy and how to integrate them through the Kivy Language.

*Chapter 2, Graphics – the Canvas,* explains the use of the canvas and how to draw vector figures on the screen.

*Chapter 3, Widget Events – Binding Actions,* teaches how to connect the interactions of the user through the interface with particular code inside the program.

*Chapter 4, Improving the User Experience,* introduces a collection of useful components to enrich the interaction of the user with the interface.

*Chapter 5, Invaders Revenge – an Interactive Multi-touch Game,* presents components and strategies to build highly interactive applications.

*Chapter 6, Kivy Player – a TED Video Streamer,* builds a responsive and professional-looking interface to control a video stream service.
Kivy is a free, open source Python library that allows for quick and easy development of highly interactive multiplatform applications. Kivy’s execution speed is comparable to the native mobile alternative, Java for Android or Objective C for iOS. Moreover, Kivy has the huge advantage of being able to run on multiple platforms, just as HTML5 does; in which case, Kivy performs better because it doesn’t rely on a heavy browser, and many of its components are implemented in C using the Cython library in such a way that most of the graphics processing runs directly in the GPU. Kivy strikes a great balance between performance and portability across various hardware and software environments. Kivy emerges with a simple but ambitious goal in mind:

"... same code for every platform, at least what we use every day: Linux/Windows/ Mac OS X/Android/iOS"

Mathieu Virbel (http://txzone.net/2011/01/kivy-next-pymt-on-android-step-1-done/)

This support has being extended to Raspberry Pi, thanks to a crowd funding campaign started by Mathieu Virbel, the creator of Kivy. Kivy was introduced for the first time at EuroPython 2011 as a Python framework designed for creating natural user interfaces. Since then, it has grown bigger and attracted an enthusiastic community.
This book requires some knowledge of Python, and very basic terminal skills, but also it requires some understanding of **Object-Oriented Programming (OOP)** concepts. In particular, it is assumed that you understand the concept of **inheritance** and the difference between **instances** and **classes**. Refer to the following table to review some of these concepts:

<table>
<thead>
<tr>
<th>Concept</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td><a href="http://en.wikipedia.org/wiki/Instance_(computer_science)">http://en.wikipedia.org/wiki/Instance_(computer_science)</a></td>
</tr>
<tr>
<td>Class</td>
<td><a href="http://en.wikipedia.org/wiki/Class_(computer_science)">http://en.wikipedia.org/wiki/Class_(computer_science)</a></td>
</tr>
</tbody>
</table>

Before we start, you will need to install Kivy. The installation process for all different platforms is documented and regularly updated on the Kivy website: [http://kivy.org/docs/installation/installation.html](http://kivy.org/docs/installation/installation.html).

All code in this book has been tested with Kivy 1.9.0 and both Python 2.7 and Python 3.4 (but 3.3 should work fine as well).

Note that packaging support for mobile is not yet complete for Python 3.3+. For now, if we want to create mobile apps for Android or iOS, we should use Python 2.7. If you want to know your Python version, you can execute `python -V` in a terminal to check your installed Python version.

In this chapter, we start by creating user interfaces using one of Kivy's most fun and powerful components – the Kivy language (.kv). The Kivy Language separates logic from presentation in order to keep an easy and intuitive code; it also links components at an interface level. In future chapters, you will also learn how to build and modify interfaces dynamically using pure Python code and Kivy as a library.
Here is a list of all the skills that you are about to learn:

- Launching a Kivy application
- Using the Kivy language
- Instantiating and personalizing widgets (GUI components) through basic properties and variables
- Differentiating between fixed, proportional, absolute, and relative coordinates
- Creating responsive GUIs through layouts
- Modularizing code in different files

This chapter covers all the basics for building a **Graphical User Interface (GUI)** in Kivy. First, we will learn techniques to run an application and how to use and integrate widgets. After that, we will introduce the main project of the book, the **Comic Creator**, and program the main structure of the GUI that we will continue using in the following two chapters. At the end of this chapter, you will be able to build a GUI starting from a pencil and paper sketch, and also learn some techniques to make the GUI responsive to the size of the window.

**Basic interface – Hello World!**

Let's put our hands on our first code.

---

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
The following is a Hello World program:

```python
# File name: hello.py
import kivy
kivy.require('1.9.0')

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

class HelloApp(App):
    def build(self):
        return Label(text='Hello World!')

if __name__=='__main__':
    HelloApp().run()
```

This is merely Python code. Launching a Kivy program is not any different from launching any other Python application.

In order to run the code, you open a terminal (line of commands or console) and specify the following command in Windows or Linux: `python hello.py --size=150x100` (`--size` is a parameter to specify the screen size).

On a Mac, you must type in `kivy` instead of `python` after installing Kivy.app in `/Applications`. Lines 2 and 3 verify that we have the appropriate version of Kivy installed on our computer.

If you try to launch our application with an older Kivy version (say 1.8.0) than the specified version, then line 3 will raise an Exception error. This Exception is not raised if we have a more recent version.

We omit the call to `kivy.require` in most of the examples in the book, but you will find it in the code that you download online (https://www.packtpub.com/), and its use is strongly encouraged in real-life projects. The program uses two classes from the Kivy library (lines 5 and 6) – `App` and `Label`. The class `App` is the starting point of any Kivy application. Consider `App` as the empty window where we will add other Kivy components.
We use the App class through inheritance; the App class becomes the base class of the HelloApp subclass or child class (line 8). In practice, this means that the HelloApp class has all the variables and methods of App, plus whatever we define in the body (lines 9 and 10) of the HelloApp class. Most importantly, App is the starting point of any Kivy application. We can see that line 13 creates an instance of HelloApp and runs it.

Now the HelloApp class's body just overrides one of the existing App class's methods, the build(self) method. This method has to return the window content. In our case, a Label that holds the text Hello World! (line 10). A Label is a widget that allows you to display some text on the screen.

A widget is a Kivy GUI component. Widgets are the minimal graphical units that we put together in order to create user interfaces.

The following screenshot shows the resulting screen after executing the hello.py code:

So, is Kivy just another library for Python? Well, yes. But as part of the library, Kivy offers its own language in order to separate the logic from the presentation and to link elements of the interface. Moreover, remember that this library will allow you to port your applications to many platforms.

Let's start to explore the Kivy language. We will separate the previous Python code into two files, one for the presentation (interface), and another for the logic. The first file includes the Python lines:

```python
14. # File name: hello2.py
15. from kivy.app import App
16. from kivy.uix.button import Label
17.
18. class Hello2App(App):
```

```python
```
GUI Basics – Building an Interface

```python
19.     def build(self):
20.         return Label()
21.
22.     if __name__=='__main__':
23.         Hello2App().run()
```

The `hello2.py` code is very similar to `hello.py`. The difference is that the method `build(self)` doesn't have the **Hello World!** message. Instead, the message has been moved to the `text` property in the Kivy language file (`hello2.kv`).

A **property** is an attribute that can be used to change the content, appearance, or behavior of a widget.

The following is the code (rules) of `hello2.kv`, which shows how we modify the `Label` content with the `text` property (line 27):

```kivy
24. # File name: hello2.kv
25. #:kivy 1.9.0
26. <Label>:
27.     text: 'Hello World!'
```

You might wonder how Python or Kivy knows that these two files (`hello2.py` and `hello2.kv`) are related. This tends to be confusing at the beginning. The key is in the name of the subclass of `App`, which in this case is `HelloApp`.

The beginning part of the `App` class's subclass name must coincide with the name of the Kivy file. For example, if the definition of the class is `class FooApp(App)`, then the name of the file has to be `foo.kv` and in the same directory of the main file (the one that executes the `run()` method of `App`).

Once that consideration is included, this example can be run in the same way we ran the previous one. We just need to be sure we are calling the main file – `python hello2.py --size=150x100`.

This is our first contact with the Kivy language, so we should have an in-depth look at it. Line 25 (`hello2.kv`) tells Python the minimal version of Kivy that should be used. It does the same thing as the previous lines 2 and 3 do in `hello.py`. The instructions that start with `#:` in the header of a Kivy language are called **directives**. We will also be omitting the version directive throughout the rest of this book, but remember to include it in your own projects.
The `<Label>`: rule (line 26) indicates that we are going to modify the `Label` class.

The Kivy language is expressed as a sequence of rules. A rule is a piece of code that defines the content, behavior, and appearance of a Kivy widget class. A rule always starts with a widget class name in angle brackets followed by a colon, like this, `<Widget Class>`:

Inside the rule, we set the `text` property with `'Hello World!'` (line 27). The code in this section will generate the same output screen as before. In general, everything in Kivy can be done using pure Python and importing the necessary classes from the Kivy library, as we did in the first example (`hello.py`). However, there are many advantages of using the Kivy language and therefore this book explains all the presentation programming in the Kivy language, unless we need to add dynamic components, in which case using Kivy as a traditional Python library is more appropriate.

If you are an experienced programmer, you might have worried that modifying the `Label` class affects all the instances we could potentially create from `Label`, and therefore they will all contain the same `Hello World` text. That is true, and we are going to study a better approach to doing this in the following section.

Basic widgets – labels and buttons

In the last section, we used the `Label` class, which is one of the multiple widgets that Kivy provides. You can think of widgets as interface blocks that we use to set up a GUI. Kivy has a complete set of widgets – buttons, labels, checkboxes, dropdowns, and many more. You can find them all in the API of Kivy under the package `kivy.uix` (http://kivy.org/docs/api-kivy.html).

We are going to learn the basics of how to create our own personalized widget without affecting the default configuration of Kivy widgets. In order to do that, we will use inheritance to create the `MyWidget` class in the `widgets.py` file:

```python
# File name: widgets.py
from kivy.app import App
from kivy.uix.widget import Widget

class MyWidget(Widget):
    pass

class WidgetsApp(App):
```
def build(self):
    return MyWidget()

if __name__=='__main__':
    WidgetsApp().run()

In line 32, we inherit from the base class Widget and create the subclass MyWidget. It is a general practice to create your own Widget for your applications instead of using the Kivy classes directly, because we want to avoid applying our changes to all future instances of the widget Kivy class. In the case of our previous example (hello2.kv), modifying the Label class (line 26) would affect all of its future instances. In line 37, we instantiated MyWidget instead of Label directly (as we did in hello2.py), so we can now distinguish between our widget (MyWidget) and the Kivy widget (Widget). The rest of the code is analogous to what we covered before.

The following is the corresponding Kivy language code (widgets.kv):

```kivy
<MyWidget>:
    Button:
        text: 'Hello'
        font_size: 32
        color: .8,.9,0,1
        pos: 0, 100
        size: 100, 50
    Button:
        text: 'World!'  
        font_size: 32
        color: .8,.9,0,1
        pos: 100,0
        size: 100, 50
```

Note that now we are using buttons instead of labels. Most of the basic widgets in Kivy work in similar ways. In fact, Button is just a subclass of Label that incorporates more properties such as background color.

Compare the notation of line 26 (<Label>:) in hello2.kv with line 43 (Button:) of the preceding code (widgets.kv). We used the rule class notation (<Class>:) for the Label (and MyWidget) class, but a different notation (Instance:) for Button. In this way, we defined that MyWidget has two instances of Button (line 43 and 49).
Finally, we set the properties of the Button instances. The **font_size** property sets the size of the text. The **color** property sets the text color and is specified in RGBA format (red, green, blue, and alpha/transparency). The properties **size** and **pos** set the size and position of the widget and consist of a pair of fixed coordinates (x for horizontal and y for vertical), the exact pixels on the window.

Note that the coordinate (0, 0) is located at the bottom-left corner, the Cartesian origin. Many other languages (including CSS) use the top-left corner as the (0, 0) coordinate, so take note!

The following screenshot shows the output of `widgets.py` and `widgets.kv` with some helpful annotations:

![Screen shot of output with annotations](image)

A couple of things can be improved in the previous code (`widgets.kv`). First, there are some repeated properties for both buttons: pos, color, and font_size. Instead of that, let's create our own Button as we did with MyWidget so it will be easy to keep the buttons' design consistent. Second, the fixed position is quite annoying because the widgets don't adjust when the screen is resized. Let's make it responsive to the screen size in the `widgets2.kv` file:

```kivy
55. # File name: widgets2.kv
56. <MyButton@Button>:
57.     color: .8,.9,0,1
58.     font_size: 32
59.     size: 100, 50
60.
61. <MyWidget>:
62.     MyButton:
63.         text: 'Hello'
64.         pos: root.x, root.top - self.height
65.     MyButton:
66.         text: 'World!'
67.         pos: root.right - self.width, root.y
```
In this code (`widgets2.kv`), we create (`<MyButton@Button>`:) and customize the `MyButton` class (lines 56 to 59) and instances (line 62 to 67). Note the differences in the manner we defined `MyWidget` and `MyButton`.

Because we did not define the `MyButton` base class in `widgets.py` as we did with `MyWidget` (line 32 of `widgets.py`), we have to specify `@Class` in the Kivy language rule (line 56). In the `MyWidget` class case, we also needed to define its class from the Python side because we instantiated it directly (line 37 of `widgets.py`).

In this example, each `Button` class's position is responsive in the sense that they are always in the corners of the screen, no matter what the window size is. In order to achieve that, we need to use two internal variables – `self` and `root`. You might be familiar with the variable `self`. As you have probably guessed, it is just a reference to the `Widget` itself. For example, `self.height` (line 64) has a value of 50 because that is the height of that particular `MyButton` class. The variable `root` is a reference to the `Widget` class at the top of the hierarchy. For example, the `root.x` (line 64) has a value of 0 because that is the position in X-axis of the `MyWidget` instance created on line 37 of `widgets.py`.

`MyWidget` uses all of the window's space by default; therefore, the origin is (0, 0). The `x` and `y` and `width` and `height` are also widget properties, which we can use to disjoint `pos` and `size` respectively.

Fixed coordinates are still a laborious way to organize widgets and elements in the window. Let's move on to something smarter – layouts.

Layouts

No doubt, fixed coordinates are the most flexible way to organize elements in an n-dimensional space; however, it is very time consuming. Instead, Kivy provides a set of layouts that will facilitate the work of organizing widgets. A `Layout` is a `Widget` subclass that implements different strategies to organize embedded widgets. For example, one strategy could be organizing widgets in a grid (`GridLayout`).

Let's start with a simple `FloatLayout` example. It works in a very similar manner to the way we organize widgets directly inside of another `Widget` subclass, except that now we can use proportional coordinates ("percentages" of the total size of the window) rather than fixed coordinates (exact pixels).
That means that we won’t need the calculations we did in the previous section with `self` and `root`. Here is the Python code of an example that resembles the previous one:

```python
# File name: floatlayout.py

from kivy.app import App
from kivy.uix.floatlayout import FloatLayout

class FloatLayoutApp(App):
    def build(self):
        return FloatLayout()

if __name__=='__main__':
    FloatLayoutApp().run()
```

There is nothing really new in the preceding code (`floatlayout.py`), except the use of `FloatLayout` (line 75). The interesting parts are in the corresponding Kivy language (`floatlayout.kv`):

```kivy
# File name: floatlayout.py
<Button>:
    color: .8,.9,0,1
    font_size: 32
    size_hint: .4, .3

<FloatLayout>:
    Button:
        text: 'Hello'
        pos_hint: {'x': 0, 'top': 1}
    Button:
        text: 'World!'
        pos_hint: {'right': 1, 'y': 0}
```

In `floatlayout.kv`, we use two new properties – `size_hint` (line 83) and `pos_hint` (lines 88 and 91). They are similar to `size` and `pos` but receive proportional coordinates with values ranging from 0 to 1; (0, 0) is the bottom-left corner and (1, 1) is the top-right corner. For example, the `size_hint` property on line 83 sets the width to 40 percent of the window width and the height to 30 percent of the current window height. Something similar happens to the `pos_hint` property (lines 88 and 91 but the notation is different – a Python dictionary where the keys (for example, ‘x’ or ‘top’) indicate which part of the widget is referenced. For example, ‘x’ is the left border.
Note that we use the `top` key instead of the `y` one on line 88 and the `right` key instead of the `x` one on line 91. The `top` and `right` keys respectively reference the top and right edges of `Button`. In this case, we could have also used `x` and `y` for both axes; for example, we could have written `pos_hint: {'x': .85, 'y': 0}` as line 91. However, the `right` and `top` keys avoid us some calculations, making the code clearer.

The next screenshot shows the result, and the available keys for the `pos_hint` dictionary:

![Diagram showing the available `pos_hint` keys](image)

The available `pos_hint` keys (`x`, `center_x`, `right`, `y`, `center_y`, and `top`) are useful to align the edges or for centering. For example, `pos_hint: {'center_x':.5, 'center_y':.5}` would align a widget in the middle no matter the size of the window.

We could have used the `top` and `right` properties with the fixed positioning of `widgets2.kv` (line 64 and 67), but note that `pos` doesn't accept Python dictionaries `({'x':0,'y':0})`, just pairs of values exclusively corresponding to `(x, y)`. Therefore, instead of using the `pos` property, we should use the `x`, `center_x`, `right`, `y`, `center_y`, and `top` properties directly (not dictionary keys). For example, instead of `pos: root.x, root.top - self.height` (line 64), we should have used:

```python
x: 0
top: root.height
```

The properties `x`, `center_x`, `right`, `y`, `center_y`, and `top` always specify **fixed coordinates** (pixels), and not proportional ones. If we want to use **proportional coordinates**, we have to be inside a `Layout` (or an `App`) and use the `pos_hint` property.
We can also force a Layout to use fixed values, but there can be conflicts if we are not careful with the properties. If we use any Layout; pos_hint and size_hint take priority. If we want to use fixed positioning properties (pos, x, center_x, right, y, center_y, top), we have to ensure that we are not using the pos_hint property. Second, if we want to use the size, height, or width properties, then we need to set a None value to the size_hint axis we want to use with absolute values. For example, size_hint: (None, .10) allows us to use height property, but it keeps the width of 10 percent for the window's size.

The following table summarizes what we have seen about the positioning and sizing properties. The first and second columns indicate the name of the property and its respective value. The third and fourth column indicate whether it is available for layouts and for widgets.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>For layouts</th>
<th>For widgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_hint</td>
<td>A pair w, h: w, and h express a proportion (from 0 to 1 or None).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>size_hint_x</td>
<td>A proportion from 0 to 1 or None, indicating width (size_hint_x) or height (size_hint_y).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>size_hint_y</td>
<td>Dictionary with one x-axis key (x, center_x, or right) and one y-axis key (y, center_y, or top). The values are proportions from 0 to 1.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>pos_hint</td>
<td>Dictionary with one x-axis key (x, center_x, or right) and one y-axis key (y, center_y, or top). The values are proportions from 0 to 1.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>size</td>
<td>A pair w, h: w and h indicating fixed width and height in pixels.</td>
<td>Yes, but set size_hint: (None, None)</td>
<td>Yes</td>
</tr>
<tr>
<td>width</td>
<td>Fixed number of pixels.</td>
<td>Yes, but set size_hint_x: None</td>
<td>Yes</td>
</tr>
<tr>
<td>height</td>
<td>Fixed number of pixels.</td>
<td>Yes, but set size_hint_y: None</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Property | Value | For layouts | For widgets
--- | --- | --- | ---
pos | A pair \((x, y)\) indicating a fixed coordinate \((x, y)\) in pixels. | Yes, but don't use \pos_hint\ | Yes
\(x, \text{right or center}_x\) | Fixed number of pixels. | Yes, but don't use \(x, \text{right or center}_x\) in \pos_hint\ | Yes
\(y, \text{top or center}_y\) | Fixed number of pixels. | Yes, but don't use \(y, \text{top or center}_y\) in \pos_hint\ | Yes

We have to be careful because some of the properties behave differently depending on the layout we are using. Kivy currently has eight different layouts, which are described in the following table. The left-hand side column shows the name of the Kivy layout class. The right-hand side column describes briefly how they work.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>FloatLayout</td>
<td>Organizes the widgets with proportional coordinates by the \size_hint\ and \pos_hint\ properties. The values are numbers between 0 and 1, indicating a proportion to the window size.</td>
</tr>
<tr>
<td>RelativeLayout</td>
<td>Operates in the same way that FloatLayout does, but the positioning properties ((\pos, \text{x, center}_x, \text{right, y, center}_y, \text{top})) are relative to the Layout size and not the window size.</td>
</tr>
<tr>
<td>GridLayout</td>
<td>Organizes widgets in a grid. You have to specify at least one of two properties - \cols\ (for columns) or \rows\ (for rows).</td>
</tr>
<tr>
<td>BoxLayout</td>
<td>Organizes widgets in one row or one column depending on whether the value of the \orientation\ property is \horizontal\ or \vertical.</td>
</tr>
<tr>
<td>StackLayout</td>
<td>Similar to BoxLayout, but it goes to the next row or column when it runs out of space. There is more flexibility to set the \orientation\ property. For example, (lr-bt) organizes the widgets in right-to-left, bottom-to-top order. Any combination of (lr) (left to right), (rl) (right to left), (tb) (top to bottom), and (bt) (bottom to top) is allowed.</td>
</tr>
<tr>
<td>Layout</td>
<td>Details</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ScatterLayout</td>
<td>Works in a similar manner to RelativeLayout but allows multitouch gesturing for rotating, scaling, and translating. It is slightly different in its implementation, so we will review it later on.</td>
</tr>
<tr>
<td>PageLayout</td>
<td>Stacks widgets on top of each other, creating a multipage effect that allows flipping of pages using side borders. Very often, we will use another layout to organize elements inside each of the pages, which are simply widgets.</td>
</tr>
</tbody>
</table>

The Kivy API (http://kivy.org/docs/api-kivy.html) offers a detailed explanation and good examples of each of the layouts. The behavioral difference of the properties depends on the layout, and it is sometimes unexpected. Here are some hints that will help us in the GUI building process:

- **size_hint, size_hint_x, and size_hint_y** work on all the layouts (except PageLayout), but the behavior might be different. For example, GridLayout will try to take an average of the x hints and y hints on the same row or column respectively.
- You should use values from 0 to 1 with size_hint, size_hint_x, and size_hint_y. However, you can use values higher than 1. Depending on the layout, Kivy makes the widget bigger than the container or tries to recalculate a proportion based on the sum of the hints on the same axis.
- **pos_hint** only works for FloatLayout, RelativeLayout, and BoxLayout. In BoxLayout, only the axis-x keys (x, center_x, right) work in the vertical orientation and vice-versa for the horizontal orientation. An analogous rule applies for the fixed positioning properties (pos, x, center_x, right, y, center_y, and top).
- **size_hint, size_hint_x, and size_hint_y** can always be set as None in favor of size, width, and height.

There are more properties and particularities of each layout, but with the ones covered, we will be able to build almost any GUI. In general, the recommendation is to use the layout as it is and, instead of forcing it with the properties we are using, it is better to have more layouts and combine them to reach our goals. The next section will teach us how to embed layouts and will offer more comprehensive examples.
Embedding layouts

Layouts are subclasses of widgets. We have already been embedding widgets inside widgets since the beginning (line 43) and it won't matter if the widgets we are embedding are also layouts. In this section, we will work with a comprehensive example to explore the effect of the position properties discussed in the previous section. The example is not visually appealing, but it will be useful to illustrate some concepts and to provide some code that you can use to test different properties. The following is the Python code (`layouts.py`) for the example:

```python
92. # File name: layouts.py
93. from kivy.app import App
94. from kivy.uix.gridlayout import GridLayout
95.
96. class MyGridLayout(GridLayout):
97.     pass
98.
99. class LayoutsApp(App):
100.     def build(self):
101.         return MyGridLayout()
102.
103.     if __name__ == '__main__':
104.         LayoutsApp().run()
```

Nothing new in the preceding code – we just created `MyGridLayout`. The final output is shown in the next screenshot, with some indications about the different layouts:
In this screenshot, six different Kivy layouts are embedded into a `GridLayout` of two rows (line 107) in order to show the behavior of different widget properties. The code is straightforward, although extensive. Therefore, we are going to study the corresponding Kivy language code (`layouts.kv`) in five fragments. The following is fragment 1:

```python
105. # File name: layouts.kv (Fragment 1)
106. <MyGridLayout>:
107.     rows: 2
108.     FloatLayout:
109.         Button:
110.             text: 'F1'
111.             size_hint: .3, .3
112.             pos: 0, 0
113.     RelativeLayout:
114.         Button:
115.             text: 'R1'
116.             size_hint: .3, .3
117.             pos: 0, 0
```

In this code, `MyGridLayout` is defined by the number of rows with the `rows` property (line 107). Then we add the first two layouts – `FloatLayout` and `RelativeLayout` with one `Button` each. Both buttons have a defined property of `pos: 0, 0` (lines 112 and 117) but note in the previous screenshot that the `Button F1` (line 109) is in the bottom-left corner of the whole window, whereas the `Button R1` (line 114) is in the bottom-left corner of `RelativeLayout`. The reason is that the `pos` coordinates in `FloatLayout` are not relative to the position of the layout.

Note that `pos_hint` always uses relative coordinates, no matter the layout we are using. In other words, the previous example wouldn't have worked if we were using `pos_hint` instead of `pos`.

In fragment 2, one `GridLayout` is added to `MyGridLayout`:

```python
118. # File name: layouts.kv (Fragment 2)
119. GridLayout:
120.     cols: 2
121.     spacing: 10
122.     Button:
123.         text: 'G1'
```
In this case, we use the `cols` property to define two columns (line 120) and the `spacing` property to separate the internal widgets by 10 pixels from each other (line 121). Also, note in the previous screenshot that the first column is thinner than the second. We achieved this by setting the `size_hint_x` to `None` and `width` to 50 of the buttons G1 (line 122) and G3 (line 128).

In fragment 3, an `AnchorLayout` is added:

```python
# File name: layouts.kv (Fragment 3)
AnchorLayout:
    anchor_x: 'right'
    anchor_y: 'top'
    Button:
        text: 'A1'
        size_hint: .5, .5
    Button:
        text: 'A2'
        size_hint: .2, .2
```

We have specified the `anchor_x` property to right and the `anchor_y` property to top (line 134 and 135) in order to arrange elements in the top-right corner of the window as shown in the previous screenshot with both buttons (lines 136 and 139). This layout is very useful to embed other layouts inside it, for example, top menu bars or side bars.

In fragment 4, a `BoxLayout` is added:

```python
# File name: layouts.kv (Fragment 4)
BoxLayout:
    orientation: 'horizontal'
    Button:
        text: 'B1'
    Button:
```

The preceding code illustrates the use of BoxLayout with the orientation property set to horizontal. Also, the lines 149 and 150 show how to use size_hint and pos_hint to move the button B2 further up.

Finally, fragment 5 adds a StackLayout:

```
153. # File name: layouts.kv (Fragment 5)
154. StackLayout:
155.     orientation: 'rl-tb'
156.     padding: 10
157.     Button:
158.         text: 'S1'
159.         size_hint: .6, .2
160.     Button:
161.         text: 'S2'
162.         size_hint: .4, .4
163.     Button:
164.         text: 'S3'
165.         size_hint: .3, .2
166.     Button:
167.         text: 'S4'
168.         size_hint: .4, .3
```

In this case, we added four buttons of different sizes. It is important to pay attention to the previous screenshot on embedding layouts to understand the rules that we applied to organize the widgets with the orientation property set to rl-tb (right to left, top to bottom, line 155). Also note that the padding property (line 156) adds 10 pixels of space between the widgets and the border of StackLayout.

**PageLayout – swiping pages**

The PageLayout works in a different manner from other layouts. It is a dynamic layout, in the sense that it allows flipping through pages using its borders. The idea is that its components are stacked in front of each other, and we can just see the one that is on top.
The following example illustrates its use, taking advantage of the example from the previous section. The Python code (pagelayout.py) is presented here:

```python
169. # File name: pagelayout.py
170. import kivy
171.
172. from kivy.app import App
173. from kivy.uix.pagelayout import PageLayout
174.
175. class MyPageLayout(PageLayout):
176.     pass
177.
178. class PageLayoutApp(App):
179.     def build(self):
180.         return MyPageLayout()
181.
182. if __name__=='__main__':
183.     PageLayoutApp().run()
```

There is nothing new in this code except the use of the PageLayout class. For the Kivy language code (pagelayout.kv), we will study the properties of PageLayout. We have simply modified the layouts.kv studied in the previous section by changing the header of the file (lines 105 to 107), now called pagelayout.kv:

```kivy
184. # File name: pagelayout.kv
185. <Layout>:
186.     canvas:
187.         Color:
188.             rgba: 1, 1, 1, 1
189.         Rectangle:
190.             pos: self.pos
191.             size: self.size
192.
193. <MyPageLayout>:
194.     page: 3
195.     border: 120
196.     swipe_threshold: .4
197.     FloatLay...
```

All the layouts inherit from a base class called Layout. In line 185, we are modifying this base class in the same way we did earlier with the Button class (line 80).
If we want to apply changes to all the child widgets that have a common base class (such as `Layout`), we can introduce those changes in the base class. Kivy will apply the changes to all the classes that derive from it.

By default, layouts don't have a background color, which is not convenient when `PageLayout` stacks them on top of each other, because we can see the elements of the layouts on the bottom. Lines 186 to 191 will draw a white (line 187 and 188) rectangle of the size (line 190) and position (line 191) of the `Layout`. In order to do this, we need to use the `canvas`, which allows us to draw shapes directly on the screen. This topic will be explained in-depth in the next chapter (`Chapter 2, Graphics - The Canvas`). You can see the result in the following screenshot:

![Canvas Drawing Example]

If you run the code on your computer, you will notice that it will take you to the page corresponding to `AnchorLayout` in the example of the previous section. The reason is that we set the `page` property to value 3 (line 194). Counting from 0, this property tells Kivy which page to display first. The `border` property tells Kivy how wide the side borders are (for sliding to the previous or the next screen). Finally, `swipe_threshold` tells the percentage of the screen that we have to slide over, in order to change the page. The next section will use some of the layouts and properties learned so far to display a more professional screen.
Our project – Comic Creator

We now have all the necessary concepts to be able to create any interface we want. This section describes the project that we will improve on, as we go through the following three chapters – the Comic Creator. The basic idea of the project is a simple application to draw a stickman. The following screenshot is a sketch (wireframe) of the GUI we are aiming for:

We can distinguish several areas in the sketch. First, we need a drawing space (top-right) for our comics. We need a tool box (top-left) with some drawing tools to draw our figures and also some general options (second from bottom to top) – clear the screen, remove the last element, group elements, change color, and use the gestures mode. Finally, it would be useful to have a status bar (center-bottom) to provide some information to the user – number of figures and last action performed.

According to what we have learned in this chapter, there are multiple solutions to organize this screen. We will use the following:

- AnchorLayout for the tool box area in the top-left corner. Inside it will be a GridLayout of two columns for the drawing tools.
- AnchorLayout for the drawing space in the top-right corner. Inside it will be a RelativeLayout to have a relative space to draw in.
- AnchorLayout for the general options and status bar area at the bottom. Inside it will be a BoxLayout with vertical orientation to organize the general options on top of the status bar:
  - BoxLayout with horizontal orientation for the buttons of the general options.
  - BoxLayout with horizontal orientation for the labels of the status bar.
We are going to use that structure by creating different files for each area—comiccreator.py, comiccreator.kv, toolbox.kv, generaltools.kv, drawingspace.kv, and statusbar.kv. Let’s start with comiccreator.py:

```python
198. # File name: comiccreator.py
199. from kivy.app import App
200. from kivy.lang import Builder
201. from kivy.uix.anchorlayout import AnchorLayout
202. Builder.load_file('toolbox.kv')
203. Builder.load_file('drawingspace.kv')
204. Builder.load_file('generaloptions.kv')
205. Builder.load_file('statusbar.kv')
206. class ComicCreator(AnchorLayout):
207.     pass
208. class ComicCreatorApp(App):
209.     def build(self):
210.         return ComicCreator()
211. if __name__ == '__main__':
212.     ComicCreatorApp().run()
```

Note that we are explicitly loading some of the files with the `Builder.load_file` instruction (lines 203 to 206). There is no need to explicitly load `comiccreator.kv` because Kivy automatically loads it by extracting the first part of the `ComicCreatorApp` name. For `ComicCreator`, we choose `AnchorLayout`. It is not the only option, but it gives clarity to the code, because the second level is also composed of `AnchorLayout` instances.

Even though using a simple widget would have been clear, it is not possible, because the `Widget` class doesn’t honor the `size_hint` and `pos_hint` properties that are necessary in the `AnchorLayout` internals.

Remember that only layouts honor the `size_hint` and `pos_hint` properties.
Here is the code for `comiccreator.kv`:

```python
216. # File name: comiccreator.kv
217. <ComicCreator>:
218.     AnchorLayout:
219.         anchor_x: 'left'
220.         anchor_y: 'top'
221.         ToolBox:
222.             id: _tool_box
223.             size_hint: None, None
224.             width: 100
225.     AnchorLayout:
226.         anchor_x: 'right'
227.         anchor_y: 'top'
228.         DrawingSpace:
229.             size_hint: None, None
230.             width: root.width - _tool_box.width
231.             height: root.height - _general_options.height - _status_bar.height
232.     AnchorLayout:
233.         anchor_x: 'center'
234.         anchor_y: 'bottom'
235.         BoxLayout:
236.             orientation: 'vertical'
237.             GeneralOptions:
238.                 id: _general_options
239.                 size_hint: 1,None
240.                 height: 48
241.             StatusBar:
242.                 id: _status_bar
243.                 size_hint: 1,None
244.                 height: 24
```
This code follows the previously presented structure for the Comic Creator. There are basically three AnchorLayout instances in the first level (lines 219, 226, and 233) and a BoxLayout that organizes the general options and the status bar (line 236).

We set the width of the ToolBox to 100 pixels and the height of the GeneralOptions and StatusBar to 48 and 24 pixels respectively (lines 241 and 245). This brings with it an interesting problem – the drawing space should use the remaining width and height of the screen (no matter the screen size). In order to achieve this, we will take advantage of the Kivy id (lines 223, 239, and 243), which allows us to refer to other components inside the Kivy language. On lines 231 and 232, we subtract tool_box.width from root.width (line 231) and general_options.height and status_bar.height from root.height (line 232).

A Kivy id allows us to create internal variables in order to access properties of other widgets inside the Kivy language rules.

For now, let’s continue exploring the Kivy language in the toolbox.kv:

245. # File name: toolbox.kv
246. <ToolButton@ToggleButton>:
247.     size_hint: None, None
248.     size: 48,48
249.     group: 'tool'
250.
251. <ToolBox@GridLayout>:
252.     cols: 2
253.     padding: 2
254.     ToolButton:
255.         text: '0'
256.     ToolButton:
257.         text: '/'
258.     ToolButton:
259.         text: '?'
We created a ToolButton class that defines the size of the drawing tools and also introduces a new Kivy widget – ToggleButton. The difference with the normal Button is that it stays pressed until we click on it again. The following is an example of the tool box with a ToolButton activated:

A ToggleButton instance can be associated with other ToggleButton instances, so just one of them is clicked on at a time. We can achieve this by assigning the same group property (line 250) to the ToggleButton instances that we want to react together. In this case, we want all the instances of ToolButton belonging to the same group, because we want to draw just one figure at a time; we make it part of the class definition (line 247).

On line 252, we implement ToolBox as a subclass of GridLayout and we add some character placeholders ('O', '/', and '?') to the ToolButton instances that we will substitute for something more appropriate in the following chapters.

The following is the code for generaloptions.kv:

```python
260. # File name: generaloptions.kv
261. <GeneralOptions@BoxLayout>:
262.     orientation: 'horizontal'
263.     padding: 2
264.     Button:
265.         text: 'Clear'
266.     Button:
267.         text: 'Remove'
268.     ToggleButton:
269.         text: 'Group'
270.     Button:
271.         text: 'Color'
272.     ToggleButton:
273.         text: 'Gestures'
```
Here is an example of how inheritance can help us separate our components. We are using `ToggleButton` instances (lines 269 and 273), and they are not affected by the previous `ToolButton` implementation. Also, we didn't associate them to any group, so they are independent of each other and will just keep a mode or state. The code only defines the `GeneralOptions` class following our initial structure. The following is the resulting screenshot:

![Screenshot of the GeneralOptions class](image)

The `statusbar.kv` file is very similar in the way it uses `BoxLayout`:

```python
text: 'Total Figures: ?'
text: "Kivy started"
```

The difference is that it organizes labels and not buttons. The following is the screenshot:

![Screenshot of the statusbar.kv file](image)

Finally, the code for `drawingspace.kv`:

```python
text: '
```
Apart from defining that `DrawingSpace` is a subclass of `RelativeLayout`, we introduce the Kivy **markup**, a nice feature for styling the text of the `Label` class. It works in a similar manner to XML-based languages. For example, in HTML, `<b>I am bold</b>` would specify bold text. First, you have to activate it (line 285) and then you just embed the text you want to style between `[tag]` and `[/tag]` (line 286). You can find the complete tag list and description in the Kivy API, in the documentation for `Label` (http://kivy.org/docs/api-kivy.uix.label.html). In the previous example, `size` and `color` are self-explanatory; `sub` refers to subindexed text; `b` to bold; and `i` to italics.

Here is the screenshot that shows the GUI of our project:

![GUI Screenshot](image)

In the following chapters, we are going to add the respective functionality to this interface that, for now, consists of placeholder widgets. However, it is exciting what we got with just a few lines of code. Our GUI is ready to go and we will be working on its logic from now on.
Chapter 1

Summary

This chapter covered all the basics and introduced some not-so-basic concepts of Kivy. We covered how to configure classes, instances, and templates. Here is a list of Kivy elements we have learned to use in this chapter:

- **Basic widgets** - `Widget`, `Button`, `ToggleButton`, and `Label`
- **Layouts** - `FloatLayout`, `RelativeLayout`, `BoxLayout`, `GridLayout`, `StackLayout`, `AnchorLayout`, and `PageLayout`
- **Properties** - `pos`, `x`, `y`, `center_x`, `center_y`, `top`, `right`, `size`, `height`, `width`, `pos_hint`, `size_hint`, `group`, `spacing`, `padding`, `color`, `text`, `font_size`, `cols`, `rows`, `orientation`, `anchor_x`, and `anchor_y`
- **Variables** - `self` and `root`
- **Others** - `id` and the markup tags `size`, `color`, `b`, `i`, and `sub`

There are many more elements from the Kivy language that we can use, but with this chapter, we have understood the general idea of how to organize elements. With the help of the Kivy API, we should be able to display most of the elements available for GUI design. There is, however, a very important element we need to study separately – the canvas, which allows us to draw vector shapes inside widgets, such as the white rectangle we draw as background in the `PageLayout` example. It is a very important topic to master in Kivy, and the entire next chapter, *Graphics - The Canvas*, will be dedicated to it.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.