Kivy Blueprints

Kivy is a Modern UI framework that greatly simplifies the development of cross-platform apps suitable for both mobile and desktop.

This book is a practical guide that will walk you through the creation of intuitive multi-platform games and apps for day-to-day use. You will learn how to build simple, common apps such as Stopwatch and Paint. Then, we will gradually delve into more advanced Python and Kivy features. We will also cover a number of related topics ranging from UI design to low-level GLSL shaders. You will be able to fill your resume with practical applications and games, including those inspired by the insanely popular puzzle game 2048 and Flappy Bird. Each chapter covers a fully functional program, highlighting different aspects of the Kivy framework.

Who this book is written for

This book is intended for programmers who are comfortable with the Python language and who want to build desktop and mobile applications with rich GUI in Python with minimal hassle. Knowledge of Kivy is not strictly required—every aspect of the framework is described when it’s first used.

What you will learn from this book

- Set up a development environment for Python and Kivy programming
- Build cross-platform applications suitable for desktop and mobile
- Create Modern UI apps reminiscent of Windows Phone flat design
- Interface with the native Android API to broaden the scope of what functionality is available to your apps
- Customize your applications by modifying the built-in Kivy features for your project specifications
- Develop full-stack, client-server solutions with the backend and UI both written in Python
- Write modular, reusable code while utilizing various aspects of the Kivy framework
- Write your own cross-platform video-games, ready for distribution on Google Play, App Store, or even Steam

Mark Vasilkov


7.50 x 9.25
330 mm x 191 mm

Community Experience Distilled
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Building a Clock App"
- A synopsis of the book’s content
- More information on Kivy Blueprints Build

About the Author

Mark Vasilkov is a software craftsman—or engineer—whichever you prefer. He specializes in Python and JavaScript development, mostly related to web and mobile applications, and has 10 years of experience in hacking stuff together so that it mostly works.

For what it's worth, Mark is a Russian Israeli. This very book was partially written in a bomb shelter due to Hamas shooting long-range rockets (containing warheads with up to 200 kg explosives each) at Tel Aviv. Israel is a beautiful country, inspiring everyone in the region to do something truly remarkable and idiosyncratic.
Kivy Blueprints

Mobile applications ceased to be the "new hotness" a long time ago, and these days users routinely expect that new software—be it a videogame or a social network—has a mobile version. Similar trend affects desktop operating systems; writing cross-platform software, once uncommon, has swiftly become a norm. Even game developers, usually limited to Microsoft operating systems on desktop, can be seen working on Mac and Linux ports for many new titles (for example, Steam, at the time of writing, hosts more than a hundred games that run on Mac and more than 50 that run on Linux).

This is especially valuable for start-ups and indie developers: building truly cross-platform software widens the potential audience, which leads to increased sales and may create good press along the way.

On the downside, writing portable software can be a very resource-hungry process, and this also affects small developers much more than big corporations.

In particular, many platforms have a preferred programming language and software development kit (SDK): iOS apps are mostly written in Objective-C and Swift, Android suggests the subpar Java programming language, and Microsoft promotes the use of the .NET framework, especially C#, for building Windows software.

Employing these tools allows you to leverage the native user interface and underlying functionality of an OS, but it also automatically prevents code reuse. This means that even if you are equally proficient in all programming languages and interfaces involved, porting the code may still take a non-trivial amount of time and introduce new bugs.

Write once, run anywhere

This whole situation creates a demand for a universal, multi-platform way to program. The problem isn't exactly new: one solution to it, created by Sun in 1995, is the Java programming language. Its marketing promise—write once, run anywhere—was never fulfilled and the language itself is unreasonably cumbersome to use. This led to many mocking variations of the slogan, culminating with write once, run away that refers to many developers abandoning Java in favor of better programming languages, including Python.

Not coincidentally, Kivy—the main topic of this book—is a graphical user interface library facilitating easy creation of multi-platform Python applications. The main features of Kivy toolkit are as follows:

- **Compatibility**: Kivy-based apps work in Linux, Mac OS X, Windows, Android, and iOS—all from a single codebase.
• **Natural user interface:** Kivy bridges the gap between different input methods, allowing you to handle a multitude of possible user interactions with similar code, mouse events and multitouch gestures alike.

• **Fast hardware-accelerated graphics:** OpenGL rendering makes Kivy suitable for creating graphics-heavy applications such as videogames, and also improves the user experience with smooth transitions.

• **The use of Python:** Kivy apps are written in Python, one of the better general purpose programming languages. In addition to being inherently portable, expressive, and readable, Python features a useful standard library and a rich ecosystem of third-party packages, the [Python Package Index](https://pypi.org), the PyPI.

Speaking of third-party packages, Kivy can be seen as a superset of many battle-tested components: a large part of its functionality relies on well-known libraries such as Pygame, SDL, and GStreamer. The API that Kivy exposes, however, is very high-level and unified.

It's worth mentioning that Kivy is free and open source MIT licensed software. In practice, this means that you can use it commercially without paying licensing fees. Its full source code is hosted on GitHub, so you can also patch bugs or add new features to it.

### What This Book Covers

*Chapter 1, Building a Clock App* provides a gentle introduction to writing applications with Kivy. It covers the Kivy language, layouts, widgets and timers. By the end of the chapter we build a simple Clock app, similar to the one found in your cellphone.

*Chapter 2, Building a Paint App* is a further exploration of the Kivy framework's components and functionality. The resulting Paint app showcases the customization of built-in widgets, drawing arbitrary shapes on canvas and handling multi-touch events.

*Chapter 3, Sound Recorder for Android* serves as an example of writing a Kivy-based Android app. It shows how to use the Pyjnius interoperability layer to load Java classes into Python, which enables us to mix Android API calls with a Kivy-based user interface.

*Chapter 4, Kivy Networking* is a hands-on guide to building a network application from the ground up. In covers a number of topics, from creating a simple protocol to writing server and client software in Python, and culminates with the Kivy Chat application.

*Chapter 5, Making a Remote Desktop App* exemplifies another way of writing client-server apps. This chapter's program is based on the HTTP protocol—the one that powers the Internet. We develop a command-line HTTP server first, and then build the Remote Desktop client app with Kivy.
Chapter 6, *Making the 2048 Game* walks you through building a playable replica of the 2048 game. We demonstrate more complex Kivy functionality, such as creating custom widgets, using Kivy properties for data binding, and processing touch screen gestures.

Chapter 7, *Writing a Flappy Bird Clone* introduces another Kivy-based game, this time it's an arcade game similar to the well-known Flappy Bird title. Over the course of this chapter we discuss the use of texture coordinates and sounds effects, implement arcade physics and collision detection.

Chapter 8, *Introducing Shaders* demonstrates the use of GLSL shaders in the context of a Kivy application. In this tutorial you will learn about OpenGL primitives such as indices and vertices, and then write incredibly fast low-level code that runs directly on the GPU.

Chapter 9, *Making a Shoot-Em-Up Game* continues where the previous chapter left off: we use the knowledge of GLSL in order to build a side-scrolling shooter. A reusable particle system class is developed along the way. This project concludes the series and capitalizes on many techniques that were explained throughout the book, such as collision detection, touch screen controls, sound effects and so on.

Appendix, *The Python Ecosystem*, gives you more on Python libraries and tools.
Building a Clock App

This book will walk you through the creation of nine little Kivy programs, each resembling a real-world use case for the Kivy framework. On many occasions, the framework will be utilized together with other Python modules fitting for the task at hand. We will see that Kivy provides a great deal of flexibility, allowing us to solve vastly different problems in a clean, concise manner.

Let's start small. In this chapter, we will build a simple Clock app, similar in concept to the built-in application found in both iOS and Android. In the first part of the chapter, we will create a non-interactive digital clock display and style it, giving our program an Android-ish flat look. We will also briefly discuss the event-driven program flow and a Kivy main loop, introducing timers used to perform recurring tasks, such as updating the screen every frame.

In the second part of this chapter, we will add a stopwatch display and controls, creating a fluid layout suitable for any screen size and orientation. A stopwatch, naturally, needs user interaction, which we are going to implement last.

The important topics introduced in this chapter are as follows:

- The basics of the Kivy language, a built-in **domain-specific language** (DSL) used to lay out widgets
- Styling (and eventually subclassing) built-in Kivy components
- Loading custom fonts and formatting text
- Scheduling and listening to events
Our finished program, depicted in the following screenshot, will only be about 60 lines long, split equally between a Python source code and a Kivy language (.kv) interface definition file.

The final look of the Clock app we're going to build.

The starting point

Our "Hello, Kivy" example from the preface is a suitable starting point for this app. We just need to add a layout container, BoxLayout, so that we can fit more than one widget on the screen later.

This is the full source code at this point:

```python
# File: main.py
from kivy.app import App

class ClockApp(App):
    pass

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

# File: clock.kv
BoxLayout:
    orientation: 'vertical'

    Label:
        text: '00:02.33'
```

[14]
Right now, it looks and behaves exactly like the previously seen "Hello, world" app. A BoxLayout container allows two or more child widgets to coexist side by side, stacking either vertically or horizontally. Given just one nested widget, as in the preceding code, BoxLayout fills up all the available screen space with it and thus becomes practically unnoticeable (it's as if Label was a root widget instead, taking over the application window). We will review layouts in more detail later on.

Note that while we may call the main.py file anything we want, the clock.kv file is autoloaded by Kivy, and therefore, has to be named after the application class. For example, if our app class is called FooBarApp, a corresponding .kv file should be named foobar.kv (the class name converted to lowercase and without the -app suffix). Closely following this naming convention allows us to avoid loading Kivy language files manually, which is unequivocally a good thing—less lines of code leading to the same result.

Modern UI

At the time of writing this, the flat design paradigm is trending in the interface design field, systematically taking over every platform, be it Web, mobile, or desktop. Prominent examples of this paradigm shift in the wild are iOS 7 and later and Windows 8 and later. Internet companies followed suit with the "Material design principles" presented at Google I/O 2014 conference, along with many other HTML5 frameworks, including the well-established ones, for example, Bootstrap.

Conveniently, the flat design emphasizes content over presentation, omitting photo-realistic shadows and detailed textures in favor of plain colors and simple geometry. It is by all means simpler to create programmatically than the "old school" skeuomorphic design that tends to be visually rich and artistic.

Skeuomorphism is a common approach to user interface design. It is characterized by applications visually imitating their real-world counterparts, for example, a Calculator app with the same button layout and look and feel as a cheap physical calculator. This may or may not help user experience (depending on who you ask).

Giving up visual details in favor of a simpler, more streamlined interface seems to be the direction everyone is going in today. On the other hand, it's naturally challenging to build a distinctive, memorable interface just from colored rectangles and such. This is why the flat design is typically synonymous with good typography; depending on the application, text is almost always a significant part of the UI, so we want it to look great.
Design inspiration
Imitation is the sincerest form of flattery, and we will imitate the clock design from Android 4.1 Jelly Bean. The distinctive feature of this design is the font weight contrast. Until it was changed in version 4.4 KitKat, the default clock used to look like this:

![Clock in Jelly Bean flavor of Android, as seen on the lock screen.](image)

The font used is Roboto, Google's Android font that superseded the Droid font family in Android 4.0 Ice Cream Sandwich.

Roboto is free for commercial use and available under the permissive Apache License. It can be downloaded from Google Fonts or from the excellent Font Squirrel library at [http://www.fontsquirrel.com/fonts/roboto](http://www.fontsquirrel.com/fonts/roboto).

Loading custom fonts
When it comes to the typography, Kivy defaults to Droid Sans—Google's earlier font. It's easy to replace Droid with a custom font, as Kivy allows us to specify the `font_name` property for textual widgets (in this case, `Label`).

In the simplest case when we have just one font variant, it is possible to assign a `.ttf` filename directly in the definition of a widget:

```python
Label:
    font_name: 'Lobster.ttf'
```

For the aforementioned design, however, we want different font weights, so this approach won't cut it. The reason being, every variation of a font (for example, bold or italic) commonly lives in a separate file, and we can only assign one filename to the `font_name` property.
Chapter 1

Our use case, involving more than one .ttf file, is better covered by a LabelBase.register static method. It accepts the following arguments (all optional), exemplified by the Roboto font family:

```python
# In Python code
LabelBase.register(name="Roboto",
                  fn_regular="Roboto-Regular.ttf",
                  fn_bold="Roboto-Bold.ttf",
                  fn_italic="Roboto-Italic.ttf",
                  fn_bolditalic="Roboto-BoldItalic.ttf")
```

After this invocation, it becomes possible to set the font_name property of a widget to the name of the previously registered font family, Roboto in this case.

This approach has two limitations to be aware of:

- Kivy only accepts TrueType .ttf font files. If the fonts are packaged as OpenType .otf or a web font format such as .woff, you may need to convert them first. This can be easily done using the FontForge editor, which can be found at http://fontforge.org/.

- There is a maximum of four possible styles per font: normal, italic, bold, and bold italic. It's fine for older font families, such as Droid Sans, but many modern fonts include anywhere from 4 to over 20 styles with varying font weight and other features. Roboto, which we're going to use shortly, is available in at least 12 styles.

<table>
<thead>
<tr>
<th>Roboto Thin</th>
<th>Roboto Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roboto Light</td>
<td>Roboto Bold</td>
</tr>
<tr>
<td>Roboto Normal</td>
<td>Roboto Ultra-Bold</td>
</tr>
</tbody>
</table>

The six font weights of Roboto font

The second point forces us to choose which font styles we will use in our application as we can't just throw in all 12, which is a bad idea anyway as it would lead to a hefty increase in file size, up to 1.7 megabytes in the case of Roboto family.

For this particular app, we only need two styles: a lighter one (Roboto-Thin.ttf) and a heavier one (Roboto-Medium.ttf), which we assign to fn_regular and fn_bold respectively:

```python
from kivy.core.text import LabelBase

LabelBase.register(name='Roboto',
                    fn_regular='Roboto-Thin.ttf',
                    fn_bold='Roboto-Medium.ttf')
```
Building a Clock App

This code should be placed right after the `__name__ == '__main__'` line in main.py, as it needs to run before the interface is created from the Kivy language definition. By the time the app class is instantiated, it might already be too late to perform basic initialization like this. This is why we have to do it in advance.

Now that we have a custom font in place, all that's left is to assign it to our `Label` widget. This can be done with the help of the following code:

```python
# In clock.kv
Label:
    text: '00:00:00'
    font_name: 'Roboto'
    font_size: 60
```

Formatting text

The most popular and universally used markup language out there is undoubtedly HTML. Kivy, on the other hand, implements a variant of BBCode, a markup language once used to format posts on many message boards. Visible distinction from HTML is that BBCode uses square brackets as tag delimiters.

The following tags are available in Kivy:

<table>
<thead>
<tr>
<th>BBCode tag</th>
<th>Effect on text</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[b]...[/b]</code></td>
<td>Bold</td>
</tr>
<tr>
<td><code>[i]...[/i]</code></td>
<td>Italic</td>
</tr>
<tr>
<td><code>[font=Lobster]...[/font]</code></td>
<td>Change font</td>
</tr>
<tr>
<td><code>[color=#FF0000]...[/color]</code></td>
<td>Set color with CSS-like syntax</td>
</tr>
<tr>
<td><code>[sub]...[/sub]</code></td>
<td>Subscript (text below the line)</td>
</tr>
<tr>
<td><code>[sup]...[/sup]</code></td>
<td>Superscript (text above the line)</td>
</tr>
<tr>
<td><code>[ref=name]...[/ref]</code></td>
<td>Clickable zone, `&lt;a href=&quot;..&quot;&gt; in HTML</td>
</tr>
<tr>
<td><code>[anchor=name]</code></td>
<td>Named location, `&lt;a name=&quot;...&quot;&gt; in HTML</td>
</tr>
</tbody>
</table>

This is by no means an exhaustive reference because Kivy is under active development and has probably undergone a number of releases since this text was written, adding new features and refining the existing functionality. Please refer to the Kivy documentation found on the official website (http://kivy.org) for an up-to-date reference manual.
Let's return to our project. To achieve the desired formatting (hours in bold and the rest of the text in fn_regular thin font), we can use the following code:

```python
Label:
  text: '[b]00[/b]:00:00'
  markup: True
```

Kivy's BBCode flavor works only if we also set the `markup` property of a widget to True, as shown in the preceding code. Otherwise, you will literally see the string `[b]...[/b]` displayed on the screen, and that's clearly not desired.

Note that if we wanted to make the whole text bold, there is no need to enclose everything in `[b]...[/b]` tags; we could just set the `bold` property of the widget to True. The same applies to italic, color, font name, and size—pretty much everything can be configured globally to affect the whole widget without touching markup.

### Changing the background color

In this section, we will adjust the window background color. Window background (the "clear color" of OpenGL renderer) is a property of a global `Window` object. In order to change it, we add this code right after the `__name__ == '__main__'` line in `main.py`:

```python
from kivy.core.window import Window
from kivy.utils import get_color_from_hex

Window.clearcolor = get_color_from_hex('#101216')
```

The `get_color_from_hex` function is not strictly required, but it's nice as it allows us to use CSS-style (#RRGGBB) colors instead of (R, G, B) tuples throughout our code. And using CSS colors is preferable for at least the following two reasons:

- **Less cognitive overhead when reading**: The `#FF0080` value is immediately recognized as a color when you're familiar with this notation, while `(255, 0, 128)` is just a bunch of numbers that may be used differently depending on the context. The floating-point variant of `#FF0080`, `(1.0, 0.0, 0.50196)` is even worse.

- **Simple and unambiguous searching**: A tuple can be arbitrarily formatted, while a CSS-like color notation is uniform, albeit case-insensitive. Performing a case-insensitive search in most text editors is very simple, as opposed to locating all instances of a given tuple inside a lengthy Python listing. The latter task can prove challenging and involve regular expressions, among other things, because the formatting of tuples doesn't have to be consistent.

We will talk more about design-related features of Kivy later on. Meanwhile, let's make our application actually show the time.

**Making the clock tick**

UI frameworks are mostly event-driven, and Kivy is no exception. The distinction from the "usual" procedural code is simple—the event-driven code needs to return to the main loop often; otherwise, it will be unable to process events from a user (such as pointer movement, clicks, or window resize), and the interface will "freeze." If you're a longtime Microsoft Windows user, you are probably familiar with programs that are unresponsive and freeze very often. It is crucial to never let this happen in our apps.

Practically, this means that we can't just code an infinite loop like this in our program:

```python
# Don't do this
while True:
    update_time()  # some function that displays time
    sleep(1)
```

Technically, it might work, but the application's UI will stay in the "not responding" state until the application gets killed (forcefully stopped) by the user or an operating system. Instead of taking this faulty approach, we need to keep in mind that there is a main loop running inside Kivy, and we need to take advantage of it by utilizing events and timers.

Event-driven architecture also means that in many places, we will listen to events to respond to various conditions, be it user input, network events, or timeouts.

One of the common events that many programs listen to is App.on_start. A method with this name, if defined on the application class, will be called as soon as the app is fully initialized. Another good example of an event that we will find in many programs is on_press, which fires when the user clicks, taps, or otherwise interacts with a button.
Speaking of time and timers, we can easily schedule our code to run in the future using a built-in `Clock` class. It exposes the following static methods:

- `Clock.schedule_once`: Runs a function once after a timeout
- `Clock.schedule_interval`: Runs a function periodically

Anyone with a JavaScript background will easily recognize these two functions. They are exactly like `window.setTimeout` and `window.setInterval` in JS. Indeed, the Kivy programming model is very similar to JavaScript even if the API looks completely different.

It's important to understand that all timed events that originate from `Clock` run as a part of Kivy's main event loop. This approach is not synonymous to threading, and scheduling a blocking function like this may prevent other events from being invoked in a timely manner, or at all.

### Updating the time on the screen

To access the `Label` widget that holds time, we will give it a unique identifier (id). Later, we can easily look up widgets based on their id property — again, a concept which is very similar to web development.

Modify `clock.kv` by adding the following:

```kv
Label:
    id: time
```

That’s it! Now we can access this `Label` widget from our code directly using the `root.ids.time` notation (`root` in our case is `BoxLayout`).

Updates to the `ClockApp` class include the addition of a method to display time, `update_time`, which looks like this:

```python
def update_time(self, nap):
    self.root.ids.time.text = strftime('[b]%H[/b]:%M:%S')
```

Now we schedule the update function to run once per second after the program starts:

```python
def on_start(self):
    Clock.schedule_interval(self.update_time, 1)
```

If we run the application right now, we'll see that the time displayed is being updated every second. To paraphrase Neil Armstrong, that is one small step for mankind, but a sizable leap for a Kivy beginner.
Building a Clock App

It's worth noting how the argument to strftime combines Kivy's BBCode-like tags described earlier with the function-specific C-style format directives. For the unfamiliar, here's a quick and incomplete reference on strftime formatting essentials:

<table>
<thead>
<tr>
<th>Format string (case-sensitive)</th>
<th>Resulting output</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S</td>
<td>Second as two digits, typically 00 to 59</td>
</tr>
<tr>
<td>%M</td>
<td>Minute as two digits, 00 to 59</td>
</tr>
<tr>
<td>%H</td>
<td>Hour as per 24-hour clock, 00 to 23</td>
</tr>
<tr>
<td>%I</td>
<td>Hour as per 12-hour clock, 01 to 12</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month, 01 to 31</td>
</tr>
<tr>
<td>%m</td>
<td>Month (numeric), 01 to 12</td>
</tr>
<tr>
<td>%B</td>
<td>Month (string), for example, &quot;October&quot;</td>
</tr>
<tr>
<td>%Y</td>
<td>Year as four digits, such as 2016</td>
</tr>
</tbody>
</table>

For the most complete and up-to-date documentation on displaying time, please refer to the official reference manual—in this case, Python standard library reference, located at https://docs.python.org/.

Binding widgets using properties

Instead of hardcoding an ID for each widget that we need to access from Python code, we can also create a property and assign it in a Kivy language file. The motivation for doing so is mostly the DRY principle and cleaner naming, at a cost of a few more lines of code.

Such a property can be defined as follows:

```python
# In main.py
from kivy.properties import ObjectProperty
from kivy.uix.boxlayout import BoxLayout

class ClockLayout(BoxLayout):
    time_prop = ObjectProperty(None)
```

In this code fragment, we make a new root widget class for our application based on BoxLayout. It has a custom property, `time_prop`, which is going to reference Label we need to address from Python code.
Additionally, in the Kivy language file, `clock.kv`, we have to bind this property to a corresponding id. Custom properties look and behave no different from the default ones and use exactly the same syntax:

```python
ClockLayout:
    time_prop: time

Label:
    id: time
```

This code makes the `Label` widget accessible from the Python code without knowing the widget's ID, using the newly defined property, `root.time_prop.text = "demo"`.

The described approach is more portable than the previously shown one and it eliminates the need to keep widget identifiers from the Kivy language file in sync with the Python code, for example, when refactoring. Otherwise, the choice between relying on properties and accessing widgets from Python via `root.ids` is a matter of coding style.

Later in this book, we'll explore more advanced usage of Kivy properties, facilitating nearly effortless data binding.

**Layout basics**

To arrange widgets on the screen, Kivy provides a number of `Layout` classes. `Layout`, a subclass of `Widget`, serves as a container for other widgets. Every layout affects the positioning and size of its children in a unique way.

For this application, we won't need anything fancy, as the desired UI is pretty straightforward. This is what we're aiming to achieve:

![A mockup layout of the finished Clock app interface.](image)
To build this, we will use BoxLayout, which is basically a one-dimensional grid. We already have BoxLayout in our clock.kv file, but since it only has one child, it does not affect anything. A rectangular grid with one cell is really just that, a rectangle.

Kivy layouts almost always try to fill the screen, thus our application will adapt to any screen size and orientation changes automatically.

If we add another label to BoxLayout, it will take half the screen space, depending on the orientation: a vertical box layout grows from top to bottom, and horizontal from left to right.

You might have guessed that in order to create a row of buttons inside a vertical layout, we can just embed another, horizontal box layout into the first one. Layouts are widgets, so they can be nested in arbitrary and creative ways to build complex interfaces.

**Finalizing the layout**

Stacking three widgets into BoxLayout normally makes every widget a third of the available size. Since we don't want buttons to be this big compared to clock displays, we can add a height property to the horizontal (inner) BoxLayout and set its vertical size_hint property to None.

The size_hint property is a tuple of two values, affecting the widget's width and height. We will discuss the impact that size_hint has on different layouts in the next few chapters; right now, let's just say that if we want to use absolute numbers for width or height, we have to set size_hint to None accordingly; otherwise, assigning size won't work as the widget will continue to compute its own size instead of using the values that we'll provide.

After updating the clock.kv file to account for stopwatch display and controls, it should look similar to the following (note the hierarchy of the layouts):

```python
BoxLayout:
    orientation: 'vertical'

Label:
    id: time
    text: '[b]00[/b]:00:00'
    font_name: 'Roboto'
    font_size: 60
    markup: True

BoxLayout:
```
If we run the code now, we'll notice that buttons don't fill all the available space inside BoxLayout. This effect is achieved using the padding and spacing properties of the layout. Padding acts very similar to CSS, pushing children (in our case, buttons) away from the edges of the layout, while spacing controls the distance between adjacent children. Both properties default to zero, aiming at maximum widget density.

**Reducing repetition**

This layout works but has one serious problem: the code is very repetitive. Every change we may want to make has to be done in a number of places throughout the file, and it's very easy to miss one of them and thus introduce an inconsistent change.
To continue the analogy with the web platform, before CSS (Cascading Style Sheets) became universally available, style information was being written directly in tags that surround the text. It looked like this:

```html
<p><font face="Helvetica">Part 1</font></p>
<p><font face="Helvetica">Part 2</font></p>
```

Using this approach, changing any individual element's properties is easy, but adjusting the properties of the whole document's look requires an excessive amount of manual labor. If we wanted to change the font face to Times in the next version of the page, we would have to search and replace every occurrence of the word Helvetica while trying to make sure that we don't have this same word in the running text, as it may be occasionally replaced too.

With style sheets, on the other hand, we move all of the styling information to a CSS rule:

```css
p {font-family: Helvetica}
```

Now we have just one place to account for styling of every paragraph throughout the document; no more searching and replacing to change font or any other visual attribute, such as color or padding. Note that we still may slightly adjust a single element's properties:

```html
<p style="font-family: Times">Part 3</p>
```

So we haven't lost anything by implementing CSS, and there is practically no tradeoff; this explains why the adaptation of style sheets on the Internet was very fast (especially considering the scale) and overwhelmingly successful. CSS is being widely used to this day with no conceptual changes.

In Kivy, there is no need to use a different file for our aggregate styles or class rules, like it's usually done in web development. We just add to the `clock.kv` file a definition like the following, outside of `BoxLayout`:

```kivy
<Label>:
    font_name: 'Roboto'
    font_size: 60
    markup: True
```

This is a class rule; it acts similar to a CSS selector described in the previous information box. Every `Label` derives all the properties from the `<Label>` class rule. (Note the angle brackets.)
Now we can remove the font_name, font_size, and markup properties from each individual Label. As a general rule, always strive to move every repeated definition into a class. This is a well-known best practice called don't repeat yourself (DRY). Changes like the one shown in the previous code snippet may seem trivial in a toy project like this but will make our code much cleaner and more maintainable in a long run.

If we want to override a property of one of the widgets, just add it as usual. Immediate properties take precedence over those inherited from the class definition.

Keep in mind that a class definition is completely different from a widget defined in the same .kv file. While the syntax is largely the same, the class is just an abstract definition; on its own, it does not create a new widget. Thus, adding a class definition will not introduce any changes to the app if we don't use it later.

### Named classes

One obvious problem with the straightforward approach to classes described earlier is that we can only have one class named Label. As soon as we need two different sets of properties applied to the same kind of widget, we have to define our own custom classes for them. Additionally, overwriting the framework's built-in classes, such as Label or Button, may have undesired consequences throughout the application, for example, if another component is using the widget we've altered under the hood.

Fortunately, this is very simple to solve. Let's create a named class for buttons, RobotoButton:

```python
<RobotoButton@Button>:
    font_name: 'Roboto'
    font_size: 25
    bold: True
```

The part before the @ symbol designates the new class name, followed by the widget type we're extending (in Python, we would say class RobotoButton(Button): instead). The resulting class can be then used in the Kivy language instead of the generic Button class:

```python
RobotoButton:
    text: 'Start'
```

The use of class rules allows us to reduce the number of recurrent lines in the clock.kv file, and also provide a consistent way of tweaking similar widgets using class definitions. Next, let's use this feature to customize all the buttons.
Styling buttons

One of the darker corners of the flat UI paradigm is the look of clickable elements, like that of buttons; there is no universally accepted way of styling them.

For example, the Modern UI style (previously called Metro, as seen in Windows 8) is very radical, with clickable elements that look mostly like flat-colored rectangles with little or no distinctive graphical features. Other vendors, such as Apple, use vibrant gradients; there is a trend of also adding rounded corners, especially in web design since CSS3 provides a special-case syntax for just that. Subtle shadows, while considered heresy by some, aren't unheard of either.

Kivy is flexible in this regard. The framework does not impose any restrictions on visuals and provides a number of useful features to implement any design you like. One of the utilities that we will discuss next is 9-patch image scaling, which is used to style buttons and similar widgets that may have borders.

9-patch scaling

The motivation for a good scaling algorithm is simple: it's almost impossible to provide pixel-perfect graphics for every button, especially for the problematic ones that contain (varying amounts of) text. Scaling images uniformly is simple to implement but yields results that are mediocre at best, partly because of the aspect ratio distortion.

Non-uniform 9-patch scaling, on the other hand, produces uncompromising quality. The idea is to split the image into static and scalable parts. The following image is a hypothetical scalable button. The middle part (shown in yellow) is the working area, and everything else is a border:
The red zones can be stretched in one dimension, while the blue zones (corners) are always left intact. This is evident from the following screenshot:

Corners, shown in blue, are fully static and may contain virtually anything. Borders, shown in red, are scalable in one dimension (top and bottom sides can be stretched horizontally, and left and right sides can be stretched vertically). The only part of the image that will be uniformly resized is the inner rectangle, the working area, shown in yellow; it is therefore common to paint it with a flat color. It will also contain text that's assigned to the button, if any.

**Using 9-patch images**

For this tutorial, we will use a simple flat button with a 1-pixel border. We can reuse this texture for all buttons or choose a different one, for example, for the Reset button. A button texture for the normal state with flat color and 1-pixel border is shown as follows:
Building a Clock App

The corresponding texture for the pressed state—an inversion of the preceding image—is shown as follows:

Now, to apply the 9-patch magic, we need to tell Kivy the size of borders that have limited scalability, as discussed previously (the image will be scaled uniformly by default). Let's revisit the `clock.kv` file and add the following properties:

```python
<RobotoButton@Button>:
    background_normal: 'button_normal.png'
    background_down: 'button_down.png'
    border: (2, 2, 2, 2)
```

The `border` property values are ordered just like in CSS: top, right, bottom, and left (that is, clockwise starting from the top). Unlike CSS, we can't supply just one value for all sides; at least in the current Kivy version (1.8), the notation `border: 2` results in error.

Probably the shortest way of setting all the borders to the same value is the Python syntax `border: [2] * 4`, which means take a list with a single element, 2, and repeat it four times.

Also note that while the visible border is just one pixel wide, we're assigning the `border` property of customized buttons to 2. This is due to the texture-stretching behavior of the renderer: if pixel colors from both sides of the "cut line" don't match, the result will be a gradient, and we want solid color.

In the class rules overview, we mentioned that the property declared on an instance of a widget takes precedence over the class rule's property with the same name. This can be used to selectively override `background_*`, `border` or any other attribute, for example, assigning another texture while reusing the border width definition:

```python
RobotoButton:
    text: 'Reset'
    background_normal: 'red_button_normal.png'
    background_down: 'red_button_down.png'
```
Now our buttons are stylized, but they still don't do anything. The next step towards our goal is making the stopwatch work.

**Counting time**

Although both stopwatch and the regular clock ultimately just display time, they are completely different in terms of functionality. Wall clock is a strictly increasing monotonic function, while stopwatch time can be paused and reset, decreasing the counter. More practically, the difference is that the operating system readily exposes its internal wall clock to Python, both directly as a `datetime` object and transparently in the case of the `strftime()` function. The latter can be called without a `datetime` argument to format the current time, which is exactly what we need for a wall clock display.

For the task of creating a stopwatch, we will need to build our own, non-monotonic time counter first. This is easily achieved without using Python's time functions altogether, thanks to Kivy's `Clock.schedule_interval` event handler that accepts the time passed between calls as a parameter. This is just what the `nap` parameter does in the following code:

```python
def on_start(self):
    Clock.schedule_interval(self.update, 0.016)

def update(self, nap):
    pass
```

Time is measured in seconds, that is, if the app is running at 60 fps and calls our function every frame, the average nap will be $60^{-1} = 0.016(6)$.

With this parameter in place, keeping track of the time passed is simple and can be achieved with a simple increment:

```python
class ClockApp(App):
    sw_seconds = 0

    def update(self, nap):
        self.sw_seconds += nap
```

This timer we just created isn't, by definition, a stopwatch since right now, there is no way for the user to actually stop it. However, let's update the display with the incrementing time first so that we can see the effect of controls immediately when implementing them.
Formatting the time for stopwatch

For the main time display, formatting is easy because the standard library function `strftime` provides us with a number of readily available primitives to convert a `datetime` object into a readable string representation, according to the provided format string.

This function has a number of limitations:

- It only accepts Python `datetime` objects (while for the stopwatch, we only have a floating-point number of seconds passed, `sw_seconds`)
- It has no formatting directive for a decimal fraction of seconds

The former `datetime` limitation can be easily circumvented: we could cast our `sw_seconds` variable to `datetime`. But the latter deficiency makes this unnecessary, as we want to end our notation with fractions of a second (exact to 0.01 sec), so `strftime` formatting just won't cut it. Hence, we implement our own time formatting.

Computing values

First, we need to compute the necessary values: minutes, seconds, and fractions of a second. The math is easy; here's the one-liner for minutes and seconds:

```python
minutes, seconds = divmod(self.sw_seconds, 60)
```

Note the use of the `divmod` function. This is a shorthand for the following:

```python
minutes = self.sw_seconds / 60
seconds = self.sw_seconds % 60
```

While being more concise, the `divmod` version should also perform better on most Python interpreters, as it performs the division just once. On today's machines, the floating-point division is quite effective, but if we run a whole lot of such operations every frame, like in a video game or simulation, the CPU time will quickly add up.

Generally, the author tends to disagree with the oft-chanted mantra about premature optimization being evil; many bad practices that lead to choppy and substandard performance can and should be easily avoided without compromising on code quality, and not doing so is by all means premature pessimization.

Also note that both `minutes` and `seconds` values are still floating-point, so we will need to convert them to integers before we print them: `int(minutes)` and `int(seconds)`.
Now all that's left is hundredths of seconds; we can compute them like this:

```python
int(seconds * 100 % 100)
```

**Putting a stopwatch in place**

We have all the values; let's join them together. Formatting strings in Python is quite a common task, and contrary to The Zen of Python commandment that reads, "There should be one—and preferably only one—obvious way to do it" ([https://www.python.org/dev/peps/pep-0020/](https://www.python.org/dev/peps/pep-0020/)), there are several common idioms for string formatting. We will use one of the simplest, operator %, which is somewhat similar to the `sprintf()` function commonly found in other programming languages:

```python
def update_time(self, nap):
    self.sw_seconds += nap
    minutes, seconds = divmod(self.sw_seconds, 60)
    self.root.ids.stopwatch.text = (n
        '%02d:%02d.[size=40]%02d[/size]' %
        (int(minutes), int(seconds),
        int(seconds * 100 % 100)))
```

Since we have fractions of a second now, the refresh frequency of 1 fps that we used earlier isn't sufficient anymore. Let's set it to 0 instead so that our `update_time` function will be called for every frame:

```python
Clock.schedule_interval(self.update_time, 0)
```

Today, most displays run at a refresh rate of 60 fps, while our value is exact to 1/100 sec, that is, changes 100 times per second. While we could have attempted to run our function at exactly 100 fps, there is absolutely no reason to do it: for users, it isn't possible to see the difference on commonly available hardware, as the display will still update no more than 60 times per second anyway.

That said, most of the time your code should work independently of a frame rate, as it relies on the user's hardware, and there is no way to predict what machine your application will end up on. Even today's smartphones have wildly different system specs and performance, let alone laptops and desktop computers.

And that's it; if we run the application now, we'll see an incrementing counter. It lacks interactivity yet, and this will be our next target.
Stopwatch controls
Controlling the application by the means of button press events is very easy. All that we need to do for this to work is use the following code:

```python
def start_stop(self):
    self.root.ids.start_stop.text = ('Start'
        if self.sw_started else 'Stop')
    self.sw_started = not self.sw_started

def reset(self):
    if self.sw_started:
        self.root.ids.start_stop.text = 'Start'
        self.sw_started = False
        self.sw_seconds = 0
```

The first event handler is for the **Start** and **Stop** buttons. It changes the state (sw_started) and the button caption. The second handler reverts everything to the initial state.

We also need to add the state property to keep track of whether the stopwatch is running or paused:

```python
class ClockApp(App):
    sw_started = False
    sw_seconds = 0

    def update_clock(self, nap):
        if self.sw_started:
            self.sw_seconds += nap
```

We change the update_clock function so that it increments sw_seconds only if the stopwatch is started, that is, sw_started is set to True. Initially, the stopwatch isn't started.

In the `clock.kv` file, we bind these new methods to on_press events:

```python
RobotoButton:
    id: start_stop
    text: 'Start'
    on_press: app.start_stop()

RobotoButton:
    id: reset
    text: 'Reset'
    on_press: app.reset()
```
In Kivy language, we have several context-sensitive references at our disposal. They are as follows:

- **self**: This always refers to the current widget;
- **root**: This is the outermost widget of a given scope;
- **app**: This is the application class instance.

As you can see, implementing event handling for buttons isn't hard at all. At this point, our app provides interaction with the stopwatch, allowing the user to start, stop, and reset it. For the purposes of this tutorial, we're done.

**Summary**

In this chapter, we built a functional Kivy app, ready to be deployed to, for example, Google Play or another app store for public use. This requires a bit of extra work and the process of packaging is platform-specific, but the hardest part—programming—is over.

With the Clock app, we managed to showcase many areas of the Kivy application's development cycle without making the code unnecessarily lengthy or convoluted. Keeping the code short and concise is a major feature of the framework because it allows us to experiment and iterate quickly. Being able to implement new bits of functionality with very little old code getting in the way is invaluable. Kivy surely lives up to its description as a library for rapid application development.

One general principle that we will encounter throughout the book (and Kivy development at large) is that neither our program nor Kivy exist in the void; we always have the whole platform at our disposal, consisting of a rich Python standard library, a lot of other libraries available from the Python cheese shop—the Python Package Index (PyPI) located at http://pypi.python.org—and elsewhere, and the underlying operating system services.

We can also retool many web-development-oriented assets easily, reusing fonts, colors, and shapes from CSS frameworks, such as Bootstrap. And by all means take a look at Google's Material design principles—this isn't just a collection of design assets, but a complete field guide that allows us to achieve a consistent and good-looking UI without sacrificing the identity or "personality" of our application.

This is, of course, only the beginning. Many features that were briefly discussed in this chapter will be explored more in-depth later in this book.
Where to buy this book

You can buy Kivy Blueprints from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.