Human Computer Interaction
The Kivy KV design language

Prof. Andrew D. Bagdanov

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze
andrew.bagdanov AT unifi.it

October 20, 2017

Outline

Where we are, where we're going

Today: the KV language and drawing

- The KV language is another step forward in organizing application code to respect the Separation of Concerns.
- We will see how this language allows us to compactly specify the structure and widget hierarchy of our applications.

Week of 30 October

- Tuesday 31 October: The MVC Paradigm and Latest and Greatest:

  Khalad Hasan, David Ahlström, Junhyeok Kim, Pourang Irani, "AirPanes: Two-Handed Around-Device Interaction for Pane Switching on Smartphones." SIGCHI, 2017.

- Wednesday 1 November: GUI Lab II (and a mini-project)
- Friday 3 November: Color and Peripheral Vision

The KV design language

Reminder: our running example

- Just a quick reminder of our running example from last week.
- This example consists of:
    - A CounterButton class that extends Button in order to implement a counter of the number of times it has bee clicked. This widget encapsulates all counter functionality within it.
    - A ConfigPane class that extends FloatLayout in order to implement a simple list of configuration options. Extending a Layout is a good way to implement custom widget collections.
    - A ButtonPane class that extends GridLayout to implement a collection of CounterButtons.
- We saw how this design can be implemented in a few ways in Kivy.
- And we saw how events can be used to communicate information about user input and state change among application Widgets.
- And finally, how Kivy properties provide an elegant solution for organizing notifications about state changes.

- This model, or mode of design for graphical user interfaces is perfectly serviceable.
- That is: using pure code to build and maintain the running state of an application throughout its lifetime.
- However, in this programming model the structure of the graphical user interface is never made explicit.
- Even in our simple example (barely a page of code), the overall hierarchy of widgets is hidden in the logic of the code building it.
- This makes it difficult to maintain and extend.
- The root problem, as is often the case, is an adequate separation of concerns.

The main idea

- In Kivy, the designers have implemented an interface design language (called the KV language, or kvlang) that allows application programmers to:
    - Specify in a compact and explicit way the overall hierarchical structure of the graphical interface (i.e. the widget hierarchy).
    - Declare properties that are dynamically added to class instances when created.
    - Respond to events using simple expressions involving widget properties.
    - Declare dynamic widget classes that extend Kivy base classes, all within the KV design language.
    - Allows us to separate how our interfaces is composed and how it is drawn from the logic of how it works.
- This design language is the last technical step we need to put us in a position to discuss the Model-View-Controller paradigm.

Side note: KV, QML and CSS

- The KV language was largely inspired by the QT QML design language.
- In fact, if you look at some examples of QML applications, you will see that KV is a greatly reduced subset of QML.
- For anyone interested in developing applications using a "real" desktop GUI toolkit, I highly recommend QT.
- Another inspiration for the KV language was CSS.
- In fact, the KV rules that we will see today are applied much in the same way as CSS rules.

Basics: declaring a widget

- To get us started, here is a basic Hello World application defined in the KV language.
- Actually, the application is not defined, only the root widget is defined (in file `hello.kv`):

```
# Define the root widget of our application. Note: it's the root
# widget because it isn't surrounded by '<' and '>'.
Label:
    text: 'Hello world!'
    font_size: '25pt'
```

- Now, in our Python file (`hello_loaded.py`):

```python
from kivy.lang import Builder
from kivy.app import App

# Our application class: just loads the KV file to get the root widget.
class MyApp(App):
    def build(self):
        return Builder.load_file('./hello.kv')


MyApp().run()
```

Basics: loading KV files

- The Python code merely defines an empty application whose `build()` method explicitly loads the KV file.
- The KV file defines a single root widget (which is simply a `Label` in this example) which is returned by `load_file()`.
- Note we can also load our interface description from a string:

```python
from kivy.lang import Builder
from kivy.app import App

# Load root widget from string.
root = Builder.load_string('''
Label:
    text: 'Hello world!'
    font_size: '25pt'
''')

# Our application class: just loads the KV file to get the root widget.
class MyApp(App):
    def build(self):
        return root

MyApp().run()
```

Basics: implicit loading of KV files

- This is one method of explicitly loading KV files, but there is also a method of implicit loading.
- If your application class is named FooApp, Kivy will automatically look for a file named foo.kv to load as the root widget:

```python
from kivy.app import App

# Our application class: really empty, since no build() method is
# defined, it will look for 'hello.kv'.
class HelloApp(App):
    pass

HelloApp().run()
```

Basics: loading of KV files (last words)

- Kivy (and Python) give you lots of flexible ways to get KV descriptions and build widget hierarchies.
- The string method is used a lot in the Kivy examples, but I absolutely hate it.
- The implicit method is convenient, but in my opinion can lead to confusing behavior.
- There are a lot of options in the Kivy App API that allow you to explicitly set the KV file associated with the application (so that it is implicitly loaded).
- You can even set the KV directory for an application, so you don't have to clutter your application directory with `.kv` files.
- Check out `kv_directory` and `kv_file` in the App API documentation.
- In my examples I will try to be as explicit as possible about how KV files are being loaded and how they are being used.

KV: Rules and context

Rules and contexts

- The Kivy language consists of a number of constructs that can be used to define or extend widgets and hierarchies:
  - Rules: A rule is similar to a CSS rule. A rule applies to specific widgets in your widget tree and modifies them in a certain way. You can use rules to specify interactive behaviour or use them to add graphical representations of the widgets they apply to.
  - A Root Widget: You can use the language to create your entire user interface. A KV file must contain only one root widget at most.
  - Dynamic Classes: Dynamic classes let you create new widgets and rules on-the-fly, without any Python declaration.
  - Templates: Templates were used to populate parts of an application, such as styling the content of a list (e.g. icon on the left, text on the right). They are now deprecated by dynamic classes.
- We have already seen one example of a root widget in our simple `hello.kv` example.

The syntax of a KV file

- A KV file consists of zero or more rules, zero or more dynamic classes, and zero or one root widget definitions:

```
# Syntax of a rule definition. Note that several Rules can share the same
# definition (as in CSS). Note the braces: they are part of the definition.
<Rule1,Rule2>:
    # .. definitions ..

<Rule3>:
    # .. definitions ..

# Syntax for creating a root widget
RootClassName:
    # .. definitions ..

# Syntax for creating a dynamic class
<NewWidget@BaseClass>:
    # .. definitions ..
```

Rules and roots

- Rules in KV files are like CSS rules: they are applied to all instances of the classes they are defined for.
- Let's look at this example:

```
# Rule to override default font size for *all* buttons.
<Button>:
    font_size: '25pt'

# A minimally interesting root widget (note *no* <>).
BoxLayout:
    orientation: 'horizontal'
    Button:
        text: 'Foo'
    Button:
        text: 'Bar'
        font_size: '10pt'
    Button:
        text: 'Mum'
```

Applying rules (and getting root)

- When we load the KV file, all rules are applied to the corresponding classes.
- This means that all Buttons will have this new font size on creation.
- [Demonstrate with explicit Button create]

```python
from kivy.lang import Builder
from kivy.uix.button import Button
from kivy.app import App

foo = Button()

# Explicitly load the KV file (applies rules).
root = Builder.load_file('./rules.kv')

# Our application class.
class MyApp(App):
    def build(self):
        return root

MyApp().run()
```

Why rules?

- The need for rules is apparent if we imagine building a complicated interface with many buttons.

```
# Example of why rules are useful.
BoxLayout:
    orientation: 'horizontal'
    Button:
        font_size: '10pt'
        padding: (15, 10)
        text: 'Foo'
    Button:
        font_size: '10pt'
        padding: (15, 10)
        text: 'Bar'
    Button:
        font_size: '10pt'
        padding: (15, 10)
        text: 'Mum'
```

The syntax of rules

- The syntax of rules and root definitions is:

```
# With the braces it's a rule. Without them, it's a root widget.
<ClassName>:
    prop1: value1
    prop2: value2

    canvas:
        CanvasInstruction1:
            canvasprop1: value1
        CanvasInstruction2:
            canvasprop2: value2

    AnotherClass:
        prop3: value1
```

- Here `prop1` and `prop2` are the properties of ClassName and `prop3` is the property of AnotherClass.
- If the widget doesnt have a property with the given name, an `ObjectProperty` will be automatically created and added to the widget.
- `AnotherClass` will be created and added as a child of the ClassName instance.

Defining the widget hierarchy

- We can now "sketch" interface hierarchies:

```
PageLayout:
    FloatLayout:
        BoxLayout:
            orientation: 'vertical'
            size_hint: (1.0, 0.25)
            Label:
                text: 'Config option 1'
            Label:
                text: 'Config option 2'
            Button:
                text: 'Go!'
    GridLayout:
        cols: 2
        Button:
            text: 'CounterButton 1'
        Button:
            text: 'CounterButton 2'
        Button:
            text: 'CounterButton 3'
        Button:
            text: 'CounterButton 4'
```

KV: Expressions, ids and keywords

Expressions

- Before we start enriching our version of the running example using a KV file, we need a few more concepts from the KV language.
- When you specify a propertys value, the value is evaluated as a Python expression that can be static or dynamic, which means that the value can use the values of other properties using reserved keywords
- The self keyword references the current widget instance:

```
Button:
    text: 'My state is %s' % self.state
```

- The root keyword is available only in rule definitions and represents the root widget of the rule:

```
<MyWidget>:
    custom: 'Hello world'
    Button:
        text: root.custom
```

Expression keywords (continued)

- The `app` keyword always refers to your app instance. Its equivalent to a call to `kivy.app.App.get_running_app()` in Python:

```
Label:
    text: app.name
```

- The `args` keyword is available in `on_<action>` callbacks and refers to the arguments passed to the callback:

```
TextInput:
    on_focus: self.insert_text("Focus" if args[1] else "No focus")
```

- Finally, class definitions may contain `ids` which can be used as a keywords:

```
<MyWidget>:
    Button:
        id: btn1
    Button:
        text: 'The state of the other button is %s' % btn1.state
```

Valid expressions

- There are two places that accept python statements in a kv file: after a `property`, which assigns to the property the result of the expression, and after an `on_property`, which executes the statement when the property is updated.
- For properties the expression can only span a single line and must return a value.
- For `on_property`, multiple single line statements are valid including multi-line statements that escape their newline, as long as they dont add an indentation level.
- My advice: don't use KV files to define complex logic requiring elaborate, multi-line expressions.

Dynamic classes

- Up until now we have only been defining widget hierarchies in terms of existing widgets.
- Or, we have been applying rules to existing widgets.
- None of this is very interesting unless we have some mechanism to create our own widgets, and to apply rules to them, and to use them in our widget hierarchies.
- KV provides dynamic classes to do this.
- This defines a class derived from Label that implements custom behavior.

```
<MyLabel@Label>:
    my_property: 100
    on_touch_down: print 'Ouch!'
```

Back to our example

- OK, let's start improving our running example with fancier KV constructions:

```
<CounterButton@Button>:
    counter: 0
    text: 'Clicks: {}'.format(self.counter)
    on_press: self.counter += 1

PageLayout:
    FloatLayout:
        BoxLayout:
            orientation: 'vertical'
            size_hint: (1.0, 0.25)
            Label:
                text: 'Config option 1'
            Label:
                text: 'Config option 2'
            Button:
                text: 'Go!'
    GridLayout:
        cols: 2
        CounterButton:
        CounterButton:
        CounterButton:
        CounterButton:
```

And now for our ConfigPane

- We can keep going with this idea by implementing a simple `ConfigOption` widget:

```
<ConfigOption@BoxLayout>:
    name: 'undefined'
    value: 0
    orientation: 'horizontal'
    Label:
        text: root.name
    TextInput:
        multiline: False
        text: str(root.value)
        on_text_validate: root.value = int(self.text)
```

Using ConfigPane

- Our root widget now becomes:

```
PageLayout:
    FloatLayout:
        BoxLayout:
            orientation: 'vertical'
            size_hint: (1.0, 0.2)
            ConfigOption:
                id: opt_columns       # So we can get the value.
                name: 'Columns'
                value: 2
            ConfigOption:
                id: opt_buttons       # So we can get the value.
                name: 'Buttons'
                value: 8
            Button:
                text: 'Go!'
                on_press:
                    button_pane.cols = int(opt_columns.value)
                    button_pane.buttons = int(opt_columns.value)
```

What about our `ButtonPane`?

- Can we keep going and implement our ButtonPane entirely in KV? Should we?

- Here is our current `ButtonPane` (with new properties and an id):

```
GridLayout:
    id: button_pane
    buttons: 4          # Number of buttons.
    cols: 2             # Number of columns.

    # Yikes. This is still fixed at four!
    CounterButton:
    CounterButton:
    CounterButton:
    CounterButton:
```

- How can we tie the structure of this declaration to the variable number of buttons?

- There might be a way to do this in pure KV, but I couldn't figure it out.

KV: From Python to KV and back

From Python to KV

- We can define our own custom widgets in Python as usual, and then use the KV design to apply rules to them.
- Here's how we will start our new `ButtonPane` widget.
- [Comment on order importance, trigger, and Factory]

```python
# Our ButtonPane class.
class ButtonPane(GridLayout):
    # Create a trigger for us to signal reconfiguration.
    def __init__(self, **kwargs):
        super(ButtonPane, self).__init__(**kwargs)
        self.trigger_reconfig = Clock.create_trigger(self.reconfigure)
        self.trigger_reconfig()

    # This simply deletes existing buttons and recreates them.
    def reconfigure(self, dt):
        self.clear_widgets()
        for i in range(self.buttons):
            self.add_widget(Factory.CounterButton())

# Explicitly load the KV file (applies rules).
root = Builder.load_file('./running_final.kv')
```

And now over to KV

- Our KV can apply rules to it, use it in widget hierarchies, and connect its properties to other events and other properties.

```
PageLayout:
    FloatLayout:
        BoxLayout:
            orientation: 'vertical'
            size_hint: (1.0, 0.2)
            ConfigOption:
                id: opt_columns      # So we can get the value.
                name: 'Columns'
                value: 2
            ConfigOption:
                id: opt_buttons      # So we can get the value.
                name: 'Buttons'
                value: 8
    ButtonPane:
        id: button_pane
        buttons: int(opt_buttons.value)     # Number of buttons.
        cols: int(opt_columns.value)        # Number of columns.
        on_cols: self.trigger_reconfig()
        on_buttons: self.trigger_reconfig()
```

The final example

- We now have a pretty decent separation of concerns.
- Most of the "tricky" logic is in Python code, while the structure of the widget hierarchy is declared in the KV design language.
- Moreover, the static dependency structure between properties is established in the KV description as well.
- In this example we didn't apply any rules to our ButtonPane in the KV file, but we easily could have (to apply defaults, for example).

KV: Canvas

The basic drawing model

- You might have noticed that when we introduced the KV syntax for rules, there was a canvas element.
- We have been suspiciously silent about this until now.
- All Kivy widgets implement a (relatively) free-form drawing space where we can render the visual content of the widget's appearance.
- In Kivy, this is called the Canvas and it is a collection of drawing instructions.
- The instructions supported are compatible with the OpenGL ES language for embedded graphics.
- [Give Wacky demonstration]

Summary

Summary

- The KV language gives us many tools to separate the visual design of our interface from its internal logic.

- Rules allow us to apply properties to base widgets (defined by Kivy already) and to our own custom widgets.

- Value expressions in KV design files also allow us to connect widget Properties together so that we do not have to worry about explicitly handling updates.

- Dynamic classes allow us to define (entirely in KV) derived classes that have logic simple enough that they do not warrant a separate Python implementation.

- These tools a extremely powerful, and give a lot of flexibility to the application designer (and more than enough rope to hang yourself).

- Care must be taken to ensure these tools are used in ways that results in increased clarity and reusability, as opposed to increased confusion.

Homework

Homework

Exercise 9.1: application configuration

Have a look at the way configuration options are defined and managed in the Kivy App API documentation. Think about how these can be incorporated into the logic of our running example.

Exercise 9.2: KV version of `ConfigPane`

Our `ConfigPane` implementation is still just a `BoxLayout` that holds our static `ConfigOptions`. What are some of the problems with this implementation? Design a new implementation that further abstracts the desired functionality of ConfigPane into a neater (and more reusable) design and implementation.

Exercise 9.3: load and run

Much of the logic in our `running_kvN.py` files is extremely redundant (we load a KV file, and then fire up an App that uses it). Design a new script that takes the name of a KV file on the command line and handles the loading and executing the KV application in these simple examples.

Exercise 9.4: Latest and Greatest

Khalad Hasan, David Ahlström, Junhyeok Kim, Pourang Irani, "AirPanes: Two-Handed Around-Device Interaction for Pane Switching on Smartphones." SIGCHI, 2017.