Programming Guide » Kv language

Table Of Contents

Concept behind the language

As your application grow more complex, it’s common that the construction of widget trees and explicit declaration of bindings, becomes verbose and hard to maintain. The *KV* Language is a attempt to overcome these short-comings.

The *KV* language (sometimes called kvlang, or kivy language), allows you to create your widget tree in a declarative way and to bind widget properties to each other or to callbacks in a natural manner. It allows for very fast prototyping and agile changes to your UI. It also facilitates a good separation between the logic of your application and its User Interface.

How to load KV

There are two ways to load Kv code into your application:

- By name convention:

  Kivy looks for a Kv file with the same name as your App class in lowercase, minus “App” if it ends with ‘App’ e.g:

  ```
  MyApp -> my.kv
  ```

  If this file defines a *Root Widget* it will be attached to the App’s *root* attribute and used as the base of the application widget tree.

- `Builder`: You can tell Kivy to directly load a string or a file. If this string or file defines a root widget, it will be returned by the method:

  ```
  Builder.load_file('path/to/file.kv')
  ```

  or:

  ```
  Builder.load_string(kv_string)
  ```

Rule context

A Kv source constitutes of *rules*, which are used to describe the content of a Widget, you can have one *root* rule, and any number of *class* or *template* rules.

The *root* rule is declared by declaring the class of your root widget, without any indentation, followed by *:* and will be set as the *root* attribute of the App instance:

```
Widget:
```

A *class* rule, declared by the name of a widget class between *< >* and followed by *:*, defines how any instance of that class will be graphically represented:

```
<MyWidget>:
```

Rules use indentation for delimitation, as python, indentation should be of four spaces per level, like the python good practice recommendations.

There are three keywords specific to Kv language:

- *app*: always refers to the instance of your application.
- *root*: refers to the base widget/template in the current rule
- *self*: always refer to the current widget

Special syntaxes

There are two special syntaxes to define values for the whole Kv context:

To access python modules and classes from kv,

```
#:import name x.y.z
#:import isdir os.path.isdir
#:import np numpy
```

is equivalent to:

```
from x.y import z as name
from os.path import isdir
import numpy as np
```

in python.

To set a global value,

```
#:set name value
```

is equivalent to:

```
name = value
```

in python.

Instantiate children

To declare the widget has a child widget, instance of some class, just declare this child inside the rule:

```
MyRootWidget:
    BoxLayout:
        Button:
        Button:
```

The example above defines that our root widget, an instance of *MyRootWidget*, which has a child that is an instance of the `BoxLayout`. That BoxLayout further has two children, instances of the `Button` class.

A python equivalent of this code could be:

```
root = MyRootWidget()
box = BoxLayout()
box.add_widget(Button())
box.add_widget(Button())
root.add_widget(box)
```

Which you may find less nice, both to read and to write.

Of course, in python, you can pass keyword arguments to your widgets at creation to specify their behaviour. For example, to set the number of columns of a `gridlayout`, we would do:

```
grid = GridLayout(cols=3)
```

To do the same thing in kv, you can set properties of the child widget directly in the rule:

```
GridLayout:
    cols: 3
```

The value is evaluated as a python expression, and all the properties used in the expression will be observed, that means that if you had something like this in python (this assume *self* is a widget with a *data* `ListProperty`):

```
grid = GridLayout(cols=len(self.data))
self.bind(data=grid.setter('cols'))
```

To have your display updated when your data change, you can now have just:

```
GridLayout:
    cols: len(root.data)
```

> Note
> Widget names should start with upper case letters while property names should start with lower case ones. Following the PEP8 Naming Conventions is encouraged.

Event Bindings

You can bind to events in Kv using the “:” syntax, that is, associating a callback to an event:

```
Widget:
    on_size: my_callback()
```

You can pass the values dispatched by the signal using the *args* keyword:

```
TextInput:
    on_text: app.search(args[1])
```

More complex expressions can be used, like:

```
pos: self.center_x - self.texture_size[0] / 2., self.center_y - self.texture_size
```

This expression listens for a change in `center_x`, `center_y`, and `texture_size`. If one of them changes, the expression will be re-evaluated to update the `pos` field.

You can also handle `on_` events inside your kv language. For example the TextInput class has a `focus` property whose auto-generated `on_focus` event can be accessed inside the kv language like so:

```
TextInput:
    on_focus: print(args)
```

Extend canvas

Kv lang can be used to define the canvas instructions of your widget like this:

```
MyWidget:
    canvas:
        Color:
            rgba: 1, .3, .8, .5
        Line:
            points: zip(self.data.x, self.data.y)
```

And they get updated when properties values change.

Of course you can use *canvas.before* and *canvas.after*.

Referencing Widgets

In a widget tree there is often a need to access/reference other widgets. The Kv Language provides a way to do this using id's. Think of them as class level variables that can only be used in the Kv language. Consider the following:

```
<MyFirstWidget>:
    Button:
        id: f_but
    TextInput:
        text: f_but.state

<MySecondWidget>:
    Button:
        id: s_but
    TextInput:
        text: s_but.state
```

An `id` is limited in scope to the rule it is declared in, so in the code above `s_but` can not be accessed outside the `<MySecondWidget>` rule.

Warning

When assigning a value to `id`, remember that the value isn't a string. There are no quotes:

> good -> `id: value`, bad -> `id: 'value'`

An `id` is a `weakref` to the widget and not the widget itself. As a consequence, storing the `id` is not sufficient to keep the widget from being garbage collected. To demonstrate:

```
<MyWidget>:
    label_widget: label_widget
    Button:
        text: 'Add Button'
        on_press: root.add_widget(label_widget)
    Button:
        text: 'Remove Button'
        on_press: root.remove_widget(label_widget)
    Label:
        id: label_widget
        text: 'widget'
```

Although a reference to `label_widget` is stored in `MyWidget`, it is not sufficient to keep the object alive once other references have been removed because it's only a weakref. Therefore, after the remove button is clicked (which removes any direct reference to the widget) and the window is resized (which calls the garbage collector resulting in the deletion of `label_widget`), when the add button is clicked to add the widget back, a `ReferenceError: weakly-referenced object no longer exists` will be thrown.

To keep the widget alive, a direct reference to the `label_widget` widget must be kept. This is achieved using `id.__self__` or `label_widget.__self__` in this case. The correct way to do this would be:

```
<MyWidget>:
    label_widget: label_widget.__self__
```

Accessing Widgets defined inside Kv lang in your python code

Consider the code below in my.kv:

```
<MyFirstWidget>:
    # both these variables can be the same name and this doesn't lead to
    # an issue with uniqueness as the id is only accessible in kv.
    txt_inpt: txt_inpt
    Button:
        id: f_but
    TextInput:
        id: txt_inpt
        text: f_but.state
        on_text: root.check_status(f_but)
```

In myapp.py:

```
...
class MyFirstWidget(BoxLayout):

    txt_inpt = ObjectProperty(None)
    def check_status(self, btn):
        print('button state is: {state}'.format(state=btn.state))
        print('text input text is: {txt}'.format(txt=self.txt_inpt))
...
```

txt_inpt is defined as a `ObjectProperty` initialized to *None* inside the Class.

```
txt_inpt = ObjectProperty(None)
```

At this point self.txt_inpt is *None*. In Kv lang this property is updated to hold the instance of the `TextInput` referenced by the id *txt_inpt*.:

```
txt_inpt: txt_inpt
```

From this point onwards, *self.txt_inpt* holds a reference to the widget identified by the id *txt_input* and can be used anywhere in the class, as in the function *check_status*. In contrast to this method you could also just pass the *id* to the function that needs to use it, like in case of *f_but* in the code above.

There is a simpler way to access objects with *id* tags in Kv using the *ids* lookup object. You can do this as follows:

```
<Marvel>
  Label:
    id: loki
    text: 'loki: I AM YOUR GOD!'
  Button:
    id: hulk
    text: "press to smash loki"
    on_release: root.hulk_smash()
```

In your python code:

```
class Marvel(BoxLayout):

    def hulk_smash(self):
        self.ids.hulk.text = "hulk: puny god!"
        self.ids["loki"].text = "loki: >_<!!!"  # alternative syntax
```

When your kv file is parsed, kivy collects all the widgets tagged with id's and places them in this *self.ids* dictionary type property. That means you can also iterate over these widgets and access them dictionary style:

```
for key, val in self.ids.items():
    print("key={0}, val={1}".format(key, val))
```

> Note
> Although the *self.ids* method is very concise, it is generally regarded as 'best practice' to use the ObjectProperty. This creates a direct reference, provides faster access and is more explicit.

Dynamic Classes

Consider the code below:

```
<MyWidget>:
    Button:
        text: "Hello world, watch this text wrap inside the button"
        text_size: self.size
        font_size: '25sp'
        markup: True
    Button:
        text: "Even absolute is relative to itself"
        text_size: self.size
        font_size: '25sp'
        markup: True
    Button:
        text: "Repeating the same thing over and over in a comp = fail"
        text_size: self.size
        font_size: '25sp'
        markup: True
    Button:
```

Instead of having to repeat the same values for every button, we can just use a template instead, like so:

```
<MyBigButt@Button>:
    text_size: self.size
    font_size: '25sp'
    markup: True

<MyWidget>:
    MyBigButt:
        text: "Hello world, watch this text wrap inside the button"
    MyBigButt:
        text: "Even absolute is relative to itself"
    MyBigButt:
        text: "repeating the same thing over and over in a comp = fail"
    MyBigButt:
```

This class, created just by the declaration of this rule, inherits from the Button class and allows us to change default values and create bindings for all its instances without adding any new code on the Python side.

Re-using styles in multiple widgets

Consider the code below in my.kv:

```
<MyFirstWidget>:
    Button:
        on_press: root.text(txt_inpt.text)
    TextInput:
        id: txt_inpt

<MySecondWidget>:
    Button:
        on_press: root.text(txt_inpt.text)
    TextInput:
        id: txt_inpt
```

In myapp.py:

```
class MyFirstWidget(BoxLayout):

    def text(self, val):
        print('text input text is: {txt}'.format(txt=val))

class MySecondWidget(BoxLayout):

    writing = StringProperty('')

    def text(self, val):
        self.writing = val
```

Because both classes share the same .kv style, this design can be simplified if we reuse the style for both widgets. You can do this in .kv as follows. In my.kv:

```
<MyFirstWidget,MySecondWidget>:
    Button:
        on_press: root.text(txt_inpt.text)
    TextInput:
        id: txt_inpt
```

By separating the class names with a comma, all the classes listed in the declaration will have the same kv properties.

Designing with the Kivy Language

One of aims of the Kivy language is to Separate the concerns of presentation and logic. The presentation (layout) side is addressed by your kv file and the logic by your py file.

The code goes in py files

Let’s start with a little example. First, the Python file named *main.py*:

```
import kivy
kivy.require('1.0.5')

from kivy.uix.floatlayout import FloatLayout
from kivy.app import App
from kivy.properties import ObjectProperty, StringProperty


class Controller(FloatLayout):
    '''Create a controller that receives a custom widget from the kv lang file.

    Add an action to be called from the kv lang file.
    '''
    label_wid = ObjectProperty()
    info = StringProperty()

    def do_action(self):
        self.label_wid.text = 'My label after button press'
        self.info = 'New info text'
```

```
class ControllerApp(App):

    def build(self):
        return Controller(info='Hello world')


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

In this example, we are creating a Controller class with 2 properties:

- info for receving some text
- label_wid for receving the label widget

In addition, we are creating a do_action() method that will use both of these properties. It will change the info text and change text in the label_wid widget.

The layout goes in controller.kv

Executing this application without a corresponding *.kv* file will work, but nothing will be shown on the screen. This is expected, because the Controller class has no widgets in it, it's just a FloatLayout. We can create the UI around the Controller class in a file named *controller.kv*, which will be loaded when we run the ControllerApp. How this is done and what files are loaded is described in the kivy.app.App.load_kv() method.

```
#:kivy 1.0

<Controller>:
    label_wid: my_custom_label

    BoxLayout:
        orientation: 'vertical'
        padding: 20

        Button:
            text: 'My controller info is: ' + root.info
            on_press: root.do_action()

        Label:
            id: my_custom_label
            text: 'My label before button press'
```

One label and one button in a vertical BoxLayout. Seems very simple. There are 3 things going on here:

1. Using data from the Controller. As soon as the info property is changed in the controller, the expression text: 'My controller info is: ' + root.info will automatically be re-evaluated, changing the text in the Button.

2. Giving data to the Controller. The expression id: my_custom_label is assigning the created Label the id of my_custom_label. Then, using my_custom_label in the expression label_wid: my_custom_label gives the instance of that Label widget to your Controller.

3. Creating a custom callback in the `Button` using the `Controller`'s `on_press` method.

    - `root` and `self` are reserved keywords, useable anywhere. `root` represents the top widget in the rule and `self` represents the current widget.
    - You can use any id declared in the rule the same as `root` and `self`. For example, you could do this in the `on_press()`:

```
Button:
    on_press: root.do_action(); my_custom_label.font_size = 18
```

And that's that. Now when we run *main.py*, *controller.kv* will be loaded so that the `Button` and `Label` will show up and respond to our touch events.

« Graphics   Integrating with other Frameworks »