LEARNING kivy

#kivy

Table of Contents

About

Chapter 1: Getting started with kivy

Remarks

Kivy is an open source Python library for the rapid development of cross-platform user interfaces. Kivy applications can be developed for Linux, Windows, OS X, Android and iOS using the same codebase.

Graphics are rendered via OpenGL ES 2 rather than through native widgets, leading to a fairly uniform appearance across operating systems.

Developing interfaces in Kivy optionally involves the use of kvlang, a small language that supports python-like expressions and python interop. The use of kvlang can drastically simplify the development of user interface as compared to using Python exclusively.

Kivy is free to use (currently under the MIT license), and professionally backed.

Examples

Installation & Setup

Windows

There are two options how to install Kivy:

First ensure python tools are up-to-date.

```
python -m pip install --upgrade pip wheel setuptools
```

Then install the basic dependencies.

```
python -m pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew
```

Although Kivy already has providers for audio & video, GStreamer is required for more advanced stuff.

```
python -m pip install kivy.deps.gstreamer --extra-index-url
https://kivy.org/downloads/packages/simple/
```

To make it simpler, `<python>` in the following text means a path to the directory with `python.exe` file.

1. Wheel

   The wheel package provides compiled Kivy, but with removed `cython` source components, which means the core code can't be recompiled using this way. Python code, however, is editable.

The stable version of Kivy is available on pypi.

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

The latest version from the official repository is available through nightly-built wheels available on google drive. Visit the link in docs matching your python version. After a proper wheel is downloaded, rename it to match the formatting of this example and run the command.

```
python -m pip install C:\Kivy-1.9.1.dev-cp27-none-win_amd64.whl
```

2. Source

   There are more required dependencies needed to install Kivy from source than using the wheels, but the installation is more flexible.

   Create a new file in `<python>\Lib\distutils\distutils.cfg` with these lines to ensure a proper compiler will be used for the source code.

```
[build]
compiler = mingw32
```

   Then the compiler is needed. Either use some you already have installed, or download `mingwpy`. The important files such as `gcc.exe` will be located in `<python>\Scripts`.

```
python -m pip install -i https://pypi.anaconda.org/carlkl/simple mingwpy
```

   Don't forget to set environment variables to let Kivy know what providers it should use.

```
set USE_SDL2=1
set USE_GSTREAMER=1
```

   Now install the additional dependencies required for the compilation.

```
python -m pip install cython kivy.deps.glew_dev kivy.deps.sdl2_dev
python -m pip install kivy.deps.gstreamer_dev --extra-index-url
https://kivy.org/downloads/packages/simple/
```

   Check `Paths` section to ensure everything is set properly and install Kivy. Choose one of these options:

```
python -m pip install C:\master.zip
python -m pip install https://github.com/kivy/kivy/archive/master.zip
```

Paths

Kivy needs an access to the binaries from some dependencies. This means the correct folders *have to* be on the environment's `PATH` variable.

```
set PATH=<python>\Tools;<python>\Scripts;<python>\share\sdl2\bin;%PATH%
```

This way Python IDLE IDE can be included to the path with `<python>\Lib\idlelib;`. Then write `idle` into console and IDLE will be ready to use Kivy.

Simplify it

To avoid repetitive setting of environment variables either set each necessary path this way or make a batch(`.bat`) file with these lines placed into `<python>`:

```
set PATH=%~dp0;%~dp0Tools;%~dp0Scripts;%~dp0share\sdl2\bin;%~dp0Lib\idlelib;%PATH%
cmd.exe
```

To run Kivy project after installation run `cmd.exe` or the batch file and use `python <filename>.py`

installation on Ubuntu

For install kivy on ubuntu with kivy example open terminal and run following command

First add ppa

```
 sudo add-apt-repository ppa:kivy-team/kivy
```

For install kivy

```
 sudo apt-get install python-kivy
```

For install kivy examples

```
 sudo apt-get install python-kivy-example
```

Touch, grab and move

The following example creates a canvas with 2 points and 1 line in between. You will be able to move the point and the line around.

```
from kivy.app import App
from kivy.graphics import Ellipse, Line
from kivy.uix.boxlayout import BoxLayout


class CustomLayout(BoxLayout):

    def __init__(self, **kwargs):
        super(CustomLayout, self).__init__(**kwargs)

        self.canvas_edge = {}
        self.canvas_nodes = {}
        self.nodesize = [25, 25]

        self.grabbed = {}

        #declare a canvas
        with self.canvas.after:
            pass

        self.define_nodes()
        self.canvas.add(self.canvas_nodes[0])
        self.canvas.add(self.canvas_nodes[1])
        self.define_edge()
        self.canvas.add(self.canvas_edge)


    def define_nodes(self):
        """define all the node canvas elements as a list"""

        self.canvas_nodes[0] = Ellipse(
            size = self.nodesize,
            pos =  [100,100]
            )

        self.canvas_nodes[1] = Ellipse(
            size = self.nodesize,
            pos =  [200,200]
            )

    def define_edge(self):
        """define an edge canvas elements"""


        self.canvas_edge = Line(
            points =  [
                self.canvas_nodes[0].pos[0] + self.nodesize[0] / 2,
                self.canvas_nodes[0].pos[1] + self.nodesize[1] / 2,
                self.canvas_nodes[1].pos[0] + self.nodesize[0] / 2,
                self.canvas_nodes[1].pos[1] + self.nodesize[1] / 2
                ],
            joint = 'round',
            cap = 'round',
            width = 3
            )


    def on_touch_down(self, touch):

        for key, value in self.canvas_nodes.items():
            if (value.pos[0] - self.nodesize[0]) <= touch.pos[0] <= (value.pos[0] +
self.nodesize[0]):
                if (value.pos[1] - self.nodesize[1]) <= touch.pos[1] <= (value.pos[1] +
self.nodesize[1]):
                    touch.grab(self)
                    self.grabbed = self.canvas_nodes[key]
                    return True

    def on_touch_move(self, touch):

        if touch.grab_current is self:
            self.grabbed.pos = [touch.pos[0] - self.nodesize[0] / 2, touch.pos[1] -
self.nodesize[1] / 2]
            self.canvas.clear()
            self.canvas.add(self.canvas_nodes[0])
            self.canvas.add(self.canvas_nodes[1])
            self.define_edge()
            self.canvas.add(self.canvas_edge)
        else:
            # it's a normal touch
            pass

    def on_touch_up(self, touch):
        if touch.grab_current is self:
            # I receive my grabbed touch, I must ungrab it!
            touch.ungrab(self)
        else:
            # it's a normal touch
            pass

class MainApp(App):

    def build(self):
        root = CustomLayout()
        return root

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

Hello world in kivy.

The following code illustrates how to make 'hello world' app in kivy.To run this app in ios and android save it as main.py and use buildozer.

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

Builder.load_string('''
<SimpleLabel>:
    text: 'Hello World'
''')


class SimpleLabel(Label):
    pass


class SampleApp(App):
    def build(self):
        return SimpleLabel()

if __name__ == "__main__":
    SampleApp().run()
```

Simple popup example in Kivy.

The following code illustrates how to do simple popup with Kivy.

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

Builder.load_string('''
<SimpleButton>:
    on_press: self.fire_popup()
<SimplePopup>:
    id:pop
    size_hint: .4, .4
    auto_dismiss: False
    title: 'Hello world!!'
    Button:
        text: 'Click here to dismiss'
        on_press: pop.dismiss()
''')


class SimplePopup(Popup):
    pass

class SimpleButton(Button):
    text = "Fire Popup !"
    def fire_popup(self):
        pops=SimplePopup()
        pops.open()

class SampleApp(App):
    def build(self):
        return SimpleButton()

SampleApp().run()
```

RecycleView

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


items = [
    {"color":(1, 1, 1, 1), "font_size": "20sp", "text": "white",     "input_data":
["some","random","data"]},
    {"color":(.5,1, 1, 1), "font_size": "30sp", "text": "lightblue", "input_data": [1,6,3]},
    {"color":(.5,.5,1, 1), "font_size": "40sp", "text": "blue",      "input_data": [64,16,9]},
    {"color":(.5,.5,.5,1), "font_size": "70sp", "text": "gray",      "input_data":
[8766,13,6]},
    {"color":(1,.5,.5, 1), "font_size": "60sp", "text": "orange",    "input_data": [9,4,6]},
    {"color":(1, 1,.5, 1), "font_size": "50sp", "text": "yellow",    "input_data":
[852,958,123]}
]


class MyButton(Button):

    def print_data(self,data):
        print(data)
```

```
KV = '''

<MyButton>:
    on_release:
        root.print_data(self.input_data)

RecycleView:
    data: []
    viewclass: 'MyButton'
    RecycleBoxLayout:
        default_size_hint: 1, None
        orientation: 'vertical'

'''


class Test(App):
    def build(self):
        root = Builder.load_string(KV)
        root.data = [item for item in items]
        return root


Test().run()
```

Different ways to run a simple app and to interact with widgets

Most kivy apps start with this structure:

```
from kivy.app import App

class TutorialApp(App):
    def build(self):
        return
TutorialApp().run()
```

There is several way to go from here :

All the codes below (except example 1 and 3) have the same widget and similar features, but show different way to build the app.

Example 1: returning a single widget (simple Hello World App)

```
from kivy.app import App
from kivy.uix.button import Button
class TutorialApp(App):
    def build(self):
        return Button(text="Hello World!")
TutorialApp().run()
```

Example 2: returning several widgets + the button prints the label's text

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button

class TutorialApp(App):
    def build(self):
        mylayout = BoxLayout(orientation="vertical")
        mylabel = Label(text= "My App")
        mybutton =Button(text="Click me!")
        mylayout.add_widget(mylabel)
        mybutton.bind(on_press= lambda a:print(mylabel.text))
        mylayout.add_widget(mybutton)
        return mylayout
TutorialApp().run()
```

Example 3: using a class (single widget) + the button prints "My Button"

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

class Mybutton(Button):
    text="Click me!"
    on_press =lambda a : print("My Button")

class TutorialApp(App):
    def build(self):
        return Mybutton()
TutorialApp().run()
```

Example 4: it's the same as ex. 2 but it shows how to use a class

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button

class MyLayout(BoxLayout):
    #You don't need to understand these 2 lines to make it work!
    def __init__(self, **kwargs):
        super(MyLayout, self).__init__(**kwargs)

        self.orientation="vertical"
        mylabel = Label(text= "My App")
        self.add_widget(mylabel)
        mybutton =Button(text="Click me!")
        mybutton.bind(on_press= lambda a:print(mylabel.text))
        self.add_widget(mybutton)

class TutorialApp(App):
    def build(self):
        return MyLayout()
TutorialApp().run()
```

With .kv language

Example 5: the same but showing how to use kv language *within* python

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
# BoxLayout: it's in the python part, so you need to import it

from kivy.lang import Builder
Builder.load_string("""
<MyLayout>
    orientation:"vertical"
    Label: # it's in the kv part, so no need to import it
        id:mylabel
        text:"My App"
    Button:
        text: "Click me!"
        on_press: print(mylabel.text)
""")
class MyLayout(BoxLayout):
    pass
class TutorialApp(App):
    def build(self):
        return MyLayout()
TutorialApp().run()
```

**Example 6: the same with the `kv` part in a `Tutorial.kv` file **

In .py:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout

class MyLayout(BoxLayout):
    pass
class TutorialApp(App):
#the kv file name will be Tutorial (name is before the "App")
    def build(self):
        return MyLayout()
TutorialApp().run()
```

In Tutorial.kv:

```
<MyLayout> # no need to import stuff in kv!
    orientation:"vertical"
    Label:
        id:mylabel
        text:"My App"
    Button:
        text: "Click me!"
        on_press: print(mylabel.text)
```

**Example 7: link to specific kv file + a def in python receiving the label.text **

In .py:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout

class MyLayout(BoxLayout):
    def printMe(self_xx, yy):
        print(yy)
class TutorialApp(App):
    def build(self):
        self.load_kv('myapp.kv')
        return MyLayout()
TutorialApp().run()
```

In myapp.kv: orientation:"vertical" Label: id:mylabel text:"My App" Button: text: "Click me!" on_press: root.printMe(mylabel.text)

Example 8: the button prints the label's text (with a def in python using `ids`(the "IDs"))

Notice that:

- `self_xx` from example 7 is replaced by `self`

In .py:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout

class MyLayout(BoxLayout):
    def printMe(self):
        print(self.ids.mylabel.text)
class TutorialApp(App):
    def build(self):
        self.load_kv('myapp.kv')
        return MyLayout()
TutorialApp().run()
```

In myapp.kv:

```
<MyLayout>
    orientation:"vertical"
    Label:
        id:mylabel
        text:"My App"
    Button:
        text: "Click me!"
        on_press: root.printMe()
```

Example 9: the button prints the label's text (with a def in python using StringProperty)

In .py:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import StringProperty
class MyLayout(BoxLayout):
    stringProperty_mylabel= StringProperty("My App")
    def printMe(self):
        print(self.stringProperty_mylabel)

class TutorialApp(App):
    def build(self):
        return MyLayout()
TutorialApp().run()
```

In Tutorial.kv:

```
<MyLayout>
    orientation:"vertical"
    Label:
        id:mylabel
        text:root.stringProperty_mylabel
    Button:
        text: "Click me!"
        on_press: root.printMe()
```

Example 10: the button prints the label's text (with a def in python using ObjectProperty)

In .py:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import ObjectProperty
class MyLayout(BoxLayout):
    objectProperty_mylabel= ObjectProperty(None)
    def printMe(self):
        print(self.objectProperty_mylabel.text)

class TutorialApp(App):
    def build(self):
        return MyLayout()
TutorialApp().run()
```

In Tutorial.kv:

```
<MyLayout>
    orientation:"vertical"
    objectProperty_mylabel:mylabel
    Label:
        id:mylabel
        text:"My App"
    Button:
        text: "Click me!"
        on_press: root.printMe()
```

Read Getting started with kivy online: https://riptutorial.com/kivy/topic/2101/getting-started-with-kivy

Chapter 2: Property

Examples

Difference between Properties and Binding.

In brief:

- Properties make it easy to pass updates from the python side to the user interface
- Binding passes the changes that happened on the user interface to the python side.

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.lang import Builder
from kivy.properties import StringProperty
from kivy.properties import ObjectProperty
from kivy.uix.textinput import TextInput
from kivy.event import EventDispatcher
Builder.load_string("""
<CustLab1@Label>
    size_hint:0.3,1
<CustLab2@Label>
    text: "Result"
    size_hint: 0.5,1
<CustButton@Button>
    text: "+1"
    size_hint: 0.1,1
<CustTextInput@TextInput>:
    multiline: False
    size_hint:0.1,1

<Tuto_Property>:
    orientation: "vertical"
    padding:10,10
    spacing: 10
    Label:
        text: "Press the 3 button (+1) several times and then modify the number in the 
TextInput.The first counter (with StringProperty but no binding) doesn't take into account the 
change that happened in the app, but the second one does.String Property makes it easy to pass 
the update from the python side to the user interface, binding pass the changes that happened 
on the user interface to the python side. "
        text_size: self.size
        padding: 20,20

    Property_no_Binding:
    Property_with_Binding:
    Simple:

<Property_no_Binding>:
    spacing: 10
    label_ObjectProperty: result
    CustLab1:
        text: "With Property but no Binding"
    CustButton:
        on_press: root.counter_textInput_StringProperty()
    CustTextInput:
        id:textinput_id
        text: root.textInput_StringProperty
    CustLab2:
        id: result

<Property_with_Binding>:
    spacing: 10
    label_ObjectProperty: result
    CustLab1:
        text: "With Property and Binding"
    CustButton:
        on_press: root.counter_textInput_StringProperty()
    CustTextInput:
        id:textinput_id
        text: root.textInput_StringProperty
        on_text: root.textInput_StringProperty = self.text   ## this is the binding
    CustLab2:
        id: result

<Simple>
    spacing: 10
    CustLab1:
        text: "Without Property"
    CustButton:
        on_press: root.simple(textinput_id, result)
    CustTextInput:
        id:textinput_id
        text: "0"
    CustLab2:
        id: result



""")

class Property_no_Binding(BoxLayout):
    textInput_StringProperty= StringProperty("0")
    label_ObjectProperty = ObjectProperty(None)
    def counter_textInput_StringProperty(self):
        self.label_ObjectProperty.text= ("Before the counter was updated:\n\n
textinput_id.text:" + self.ids.textinput_id.text + "\n\n textInput_StringProperty:" +
self.textInput_StringProperty)
        self.textInput_StringProperty =str(int(self.textInput_StringProperty)+1)

class Property_with_Binding(BoxLayout):
    textInput_StringProperty= StringProperty("0")
    label_ObjectProperty = ObjectProperty(None)
    def counter_textInput_StringProperty(self):
        self.label_ObjectProperty.text= ("Before the counter was updated:\n\n
textinput_id.text:" + self.ids.textinput_id.text + "\n\n textInput_StringProperty:" +
self.textInput_StringProperty)
        self.textInput_StringProperty =str(int(self.textInput_StringProperty)+1)
    pass

class Simple(BoxLayout):
    def simple(self,textinput_id, result):
        result.text = ("Before the counter was updated:\n\nIn the TextInput:" +
textinput_id.text)
        textinput_id.text = str(int(textinput_id.text) + 1)
        pass
class Tuto_Property(BoxLayout):
    # def __init__(self, **kwargs):
        # super(All, self).__init__(**kwargs)
        # app=App.get_running_app()
        # self.objproper_number.bind(text=lambda *a: self.change(app))
        # print(self.parent)

    # def counter(self,app):
        # print("Stringproperty:",app.numbertext)
        # print("ObjectProperty:",self.objproper_number.text)
        # print("text:",self.ids.number.text,"\n")
        # app.numbertext=str(int(app.numbertext)+1)

    # def change(self, app):
        # app.numbertext=self.objproper_number.text
    pass
class MyApp(App):
    numbertext = StringProperty("0")
    def build(self):
        return Tuto_Property()


MyApp().run()
```

Read Property online: https://riptutorial.com/kivy/topic/9904/property

Chapter 3: Using the Screen Manager

Remarks

Circular Imports

This is a big issue in Kivy, Python, and many programming languages

When one resource is required by two files, it is normal to place this resource in the file that will be using it most. But if this happens with two resources, and they end up in opposite files, then importing both into Python will result in a circular import.

Python will import the first file, but this file imports the second. In the second, this imports the first file, which in turn imports the second and so on. Python throws the error `ImportError : cannot import name <classname>`

This can be solved by using a third file, and importing this third file into the first two. This is `resources.py` in the second example.

Examples

Simple Screen Manager Usage

```
# A line used mostly as the first one, imports App class
# that is used to get a window and launch the application
from kivy.app import App

# Casual Kivy widgets that reside in kivy.uix
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.uix.screenmanager import SlideTransition

# Inherit Screen class and make it look like
# a simple page with navigation

class CustomScreen(Screen):

    # It's necessary to initialize a widget the class inherits
    # from to access its methods such as 'add_widget' with 'super()'

    def __init__(self, **kwargs):
        # Py2/Py3 note: although in Py3 'super()' is simplified
        # it's a good practice to use Py2 syntax, so that the
        # code is compatibile in both versions
        super(CustomScreen, self).__init__(**kwargs)

        # Put a layout in the Screen which will take
        # Screen's size and pos.
        # The 'orientation' represents a direction
        # in which the widgets are added into the
        # BoxLayout - 'horizontal' is the default
        layout = BoxLayout(orientation='vertical')

        # Add a Label with the name of Screen
        # and set its size to 50px
        layout.add_widget(Label(text=self.name, font_size=50))

        # Add another layout to handle the navigation
        # and set the height of navigation to 20%
        # of the CustomScreen
        navig = BoxLayout(size_hint_y=0.2)

        # Create buttons with a custom text
        prev = Button(text='Previous')
        next = Button(text='Next')

        # Bind to 'on_release' events of Buttons
        prev.bind(on_release=self.switch_prev)
        next.bind(on_release=self.switch_next)

        # Add buttons to navigation
        # and the navigation to layout
        navig.add_widget(prev)
        navig.add_widget(next)
        layout.add_widget(navig)

        # And add the layout to the Screen
        self.add_widget(layout)

    # *args is used to catch arguments that are returned
    # when 'on_release' event is dispatched

    def switch_prev(self, *args):
        # 'self.manager' holds a reference to ScreenManager object
        # and 'ScreenManager.current' is a name of a visible Screen
        # Methods 'ScreenManager.previous()' and 'ScreenManager.next()'
        # return a string of a previous/next Screen's name
        self.manager.transition = SlideTransition(direction="right")
        self.manager.current = self.manager.previous()

    def switch_next(self, *args):
        self.manager.transition = SlideTransition(direction="right")
        self.manager.current = self.manager.next()


class ScreenManagerApp(App):

    # 'build' is a method of App used in the framework it's
    # expected that the method returns an object of a Kivy widget

    def build(self):
        # Get an object of some widget that will be the core
        # of the application - in this case ScreenManager
        root = ScreenManager()

        # Add 4 CustomScreens with name 'Screen <order>`
        for x in range(4):
            root.add_widget(CustomScreen(name='Screen %d' % x))
        # Return the object
        return root


# This is only a protection, so that if the file
# is imported it won't try to launch another App

if __name__ == '__main__':
    # And run the App with its method 'run'
    ScreenManagerApp().run()
```

Screen Manager

In the following example there are 2 Screens: SettingsScreen and MenuScreen

Using the first button, on the current screen will change your screen to the other screen.

Here is the code:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.screenmanager import ScreenManager, Screen

# Create both screens. Please note the root.manager.current: this is how
# you can control the ScreenManager from kv. Each screen has by default a
# property manager that gives you the instance of the ScreenManager used.
Builder.load_string("""
<MenuScreen>:
    BoxLayout:
        Button:
            text: 'First Button on Menu'
            on_press: root.manager.current = 'settings'
        Button:
            text: 'Second Button on Menu'

<SettingsScreen>:
    BoxLayout:
        Button:
            text: 'First Button on Settings'
            on_press: root.manager.current = 'menu'
        Button:
            text: 'Second Button on Settings'

""")

# Declare both screens
class MenuScreen(Screen):
    pass

class SettingsScreen(Screen):
    pass

# Create the screen manager
sm = ScreenManager()
sm.add_widget(MenuScreen(name='menu'))
sm.add_widget(SettingsScreen(name='settings'))
```

```
class TestApp(App):

    def build(self):
        return sm

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

Read Using the Screen Manager online: https://riptutorial.com/kivy/topic/6097/using-the-screen-manager

Credits

S. No	Chapters	Contributors
1	Getting started with kivy	Community, Daniel Engel, EL3PHANTEN, Enora, Fermi paradox, JinSnow, Kallz, KeyWeeUsr, phunsukwangdu, picibucor, user2314737, Will
2	Property	Enora, YOSHI
3	Using the Screen Manager	KeyWeeUsr, M Ganesh, OllieNye, picibucor