DW

10.009 - 2019

GUI: KIVY

NATALIE AGUS

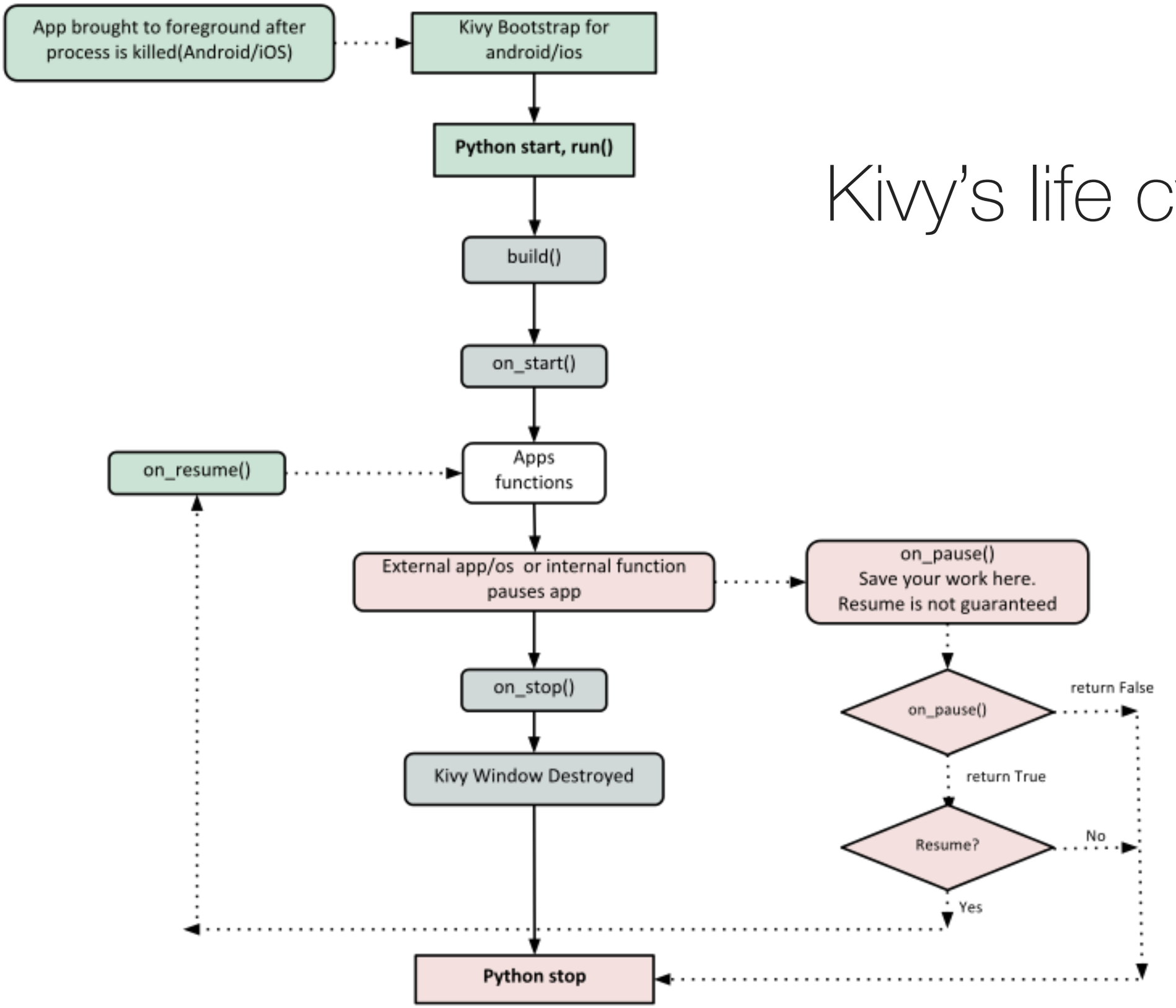
Objectives

- **Import** GUI library and create **new** GUI application
- The **main** loop
- Callback for **event**-triggered **actions**
- Create **callables**
- Create **bindings**

```python
import kivy
kivy.require('1.0.6') # replace with your current kivy version !

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


# (1) subclassing the "App" class
class MyApp(App):

    # (2) implement its "build" method
    def build(self):
        # (3) make it return a "Widget" instance
        return Label(text='Hello world')


# the main function is called when this script is run
if __name__ == '__main__':
    # instantiate the class above
    MyApp().run()
```

To create an “App”, we have to (1) inherit the Kivy’s App class (2) implement its build method by returning a (3) *root* widget.

Kivy's life cycle



TAKEN FROM : HTTPS://KIVY.ORG/DOC/STABLE/GUIDE/BASIC.HTML

```python
from kivy.app import App
from kivy.uix.gridlayout import GridLayout # widget
from kivy.uix.label import Label # widget
from kivy.uix.textinput import TextInput # widget


class LoginScreen(GridLayout):

    def __init__(self, **kwargs):
        super(LoginScreen, self).__init__(**kwargs)
        self.cols = 2
        self.add_widget(Label(text='User Name'))
        self.username = TextInput(multiline=False)
        self.add_widget(self.username)
        self.add_widget(Label(text='password'))
        self.password = TextInput(password=True, multiline=False)
        self.add_widget(self.password)


class MyApp(App):

    def build(self):
        return LoginScreen()


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

- The `GridLayout` arranges children in a matrix.
- It takes the available space and divides it into columns and rows, then adds widgets to the resulting “cells”.
- So here, we set it to have two columns, and each time we `add_widget` it will insert it to the cell accordingly
- Size auto-fitted if you drag the window

Our learned widgets so far:

- **`GridLayout:`** https://kivy.org/doc/stable/api-kivy.uix.gridlayout.html
- **`TextInput:`** https://kivy.org/doc/stable/api-kivy.uix.textinput.html
- **`Label:`** https://kivy.org/doc/stable/api-kivy.uix.label.html

What is **kwargs?

- “keyword arguments”
- like a dictionary that maps each keyword to the value that we pass to it.
- This allows us to pass variable length argument

```
def function(**kwargs):
        for key, value in kwargs.items():
              print(key, value)

function(first=“Hello”, second=“World”)
```

Interacting with your app using: **event**, **callable**, and **bindings**

- **Widget inherits `EventDispatcher`.**
- **Actually, any objects that produce events in Kivy implement the `EventDispatcher`**
- **This provides a consistent interface for registering and manipulating event handlers.**

```python
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import ListProperty

class CustomBtn(Button):
    pressed = ListProperty([0, 0])

    def on_touch_down(self, touch):
        if self.collide_point(*touch.pos):
            self.pressed = touch.pos
            return True
```

Lets create a Button (yes its another widget).

- The Widget class has many “standard” event, e.g: **`on_touch_down`**.
- Each of these “standard” event has a **`default handler`**
- We override the **`on_touch_down`** default method (handler)
- This function checks for collision of the touch with our widget.
- If the touch falls inside of our widget, we update our property **pressed**.
- This will in turn trigger **`on_pressed`** event (execute this function) automatically

```python
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import ListProperty

class CustomBtn(Button):
    pressed = ListProperty([0, 0])

    def on_touch_down(self, touch):
        if self.collide_point(*touch.pos):
            self.pressed = touch.pos
            return True

    def on_pressed(self, instance, pos):
        print ('pressed at {pos}'.format(pos=pos))
```

Lets say we also want to create our own event! Define a `property` called '`pressed`'. Kivy properties, by default, provide an `on_<property_name>` event. This event is automatically called when the value of the property is changed. So the `pressed` event will be called whenever the value of `pressed` property is changed.

```
class RootWidget(BoxLayout):
    def __init__(self, **kwargs):
        super(RootWidget, self).__init__(**kwargs)

        # create two buttons, btn 2 will print when pressed
        # btn 1 will not
        self.add_widget(Button(text='btn 1'))
        button = CustomBtn(text='btn 2')
        button.bind(pressed=self.btn_pressed)
        self.add_widget(button)

    def btn_pressed(self, instance, pos):
        print ('pos: printed from root widget: {pos}'.format(pos=pos))

class TestApp(App):
    def build(self):
        return RootWidget()


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

Now, we want to interact with our app's Button. We only have access to the root Widget instance, not the button (unless you want to have a button as your entire app, that'll be weird). We need to put the button widget inside the root widget.

```python
class RootWidget(BoxLayout):
    def __init__(self, **kwargs):
        super(RootWidget, self).__init__(**kwargs)

        # create two buttons, btn 2 will print when pressed
        # btn 1 will not
        self.add_widget(Button(text='btn 1'))
        button = CustomBtn(text='btn 2')
        button.bind(pressed=self.btn_pressed)
        self.add_widget(button)

    def btn_pressed(self, instance, pos):
        print ('pos: printed from root widget: {pos}'.format(pos=pos))

class TestApp(App):
    def build(self):
        return RootWidget()


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

We can "`bind`" a property of the child widget with a `callable` (function) within the parent widget. Do:

```
your_widget_instance.bind(property_name=function_name)
```

This means, `btn_pressed` is also called whenever `pressed` changes

Our learned widgets so far:

- **`GridLayout:`** https://kivy.org/doc/stable/api-kivy.uix.gridlayout.html
- **`TextInput:`** https://kivy.org/doc/stable/api-kivy.uix.textinput.html
- **`Label:`** https://kivy.org/doc/stable/api-kivy.uix.label.html
- **`Button`**: https://kivy.org/doc/stable/api-kivy.uix.button.html
- **`BoxLayout`**: https://kivy.org/doc/stable/api-kivy.uix.boxlayout.html

Our learned properties so far:

- **`ListProperty`**
- You should checkout other properties too
  - https://kivy.org/doc/stable/api-kivy.properties.html
- Super useful to define events and bind to them

- `StringProperty`
- `NumericProperty`
- `BoundedNumericProperty`
- `ObjectProperty`
- `DictProperty`
- `ListProperty`
- `OptionProperty`
- `AliasProperty`
- `BooleanProperty`
- `ReferenceListProperty`

What is *args?

- “non key-worded arguments”
- This also allows us to pass variable length argument, but just the values, without key

```
def function(*args):
        for value in kwargs.items():
              print(value)

function(“Hello”, “World”)
```

```python
class MyEventDispatcher(EventDispatcher):
    def __init__(self, **kwargs):
        self.register_event_type('on_test') # register
        super(MyEventDispatcher, self).__init__(**kwargs)

    def do_something(self, value):
        # (2) dispatch on_test event, this triggers any callables bind to the event
        # (3) also trigger on_test default handler
        self.dispatch('on_test', value)

    # default handler
    def on_test(self, *args):
        print ("I am dispatched", args)


class RootWidget(BoxLayout):
    def __init__(self, **kwargs):
        super(RootWidget, self).__init__(**kwargs)
        ev = MyEventDispatcher()
        # bind on_test event, with my_callback
        # means, my_callback will be executed when there's on_test event
        ev.bind(on_test=self.my_callback)
        # (1) this cause on_test event to be triggered
        ev.do_something('test')

    def my_callback(value, *args):
        print ("Hello, I got an event!")
```

```python
class TestApp(App):
    def build(self):
        return RootWidget()


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

To create an event dispatcher with `custom events`, you need to **register** the name of the event in the class and then **create** a method of the same name as **default handler.**

`You must implement a default handler, otherwise your program will return an error.`

Summary

- **Import** GUI library and create **new** GUI application
- The **main** loop, lifecycle of Kivy App

- Callback for **event**-triggered **actions using properties: ListProperty**
- Create **callables**
- Create **bindings**

- Widgets:
  - Arrange widgets in a frame:
    - **`GridLayout`**
    - **`BoxLayout`**
  - Add widgets within root widget:
    - **`Button`**
    - **`Label`**
    - **`TextInput`**