

Python for Practice



Kivy

Overview



- <https://kivy.org>
- Desktop and mobile applications
- Multitouch interface
- Multiplatform
 - Windows, Linux, Android, OS X
- Own markup language for UI design

Hello world



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

See
e01_hello.py

Application lifecycle

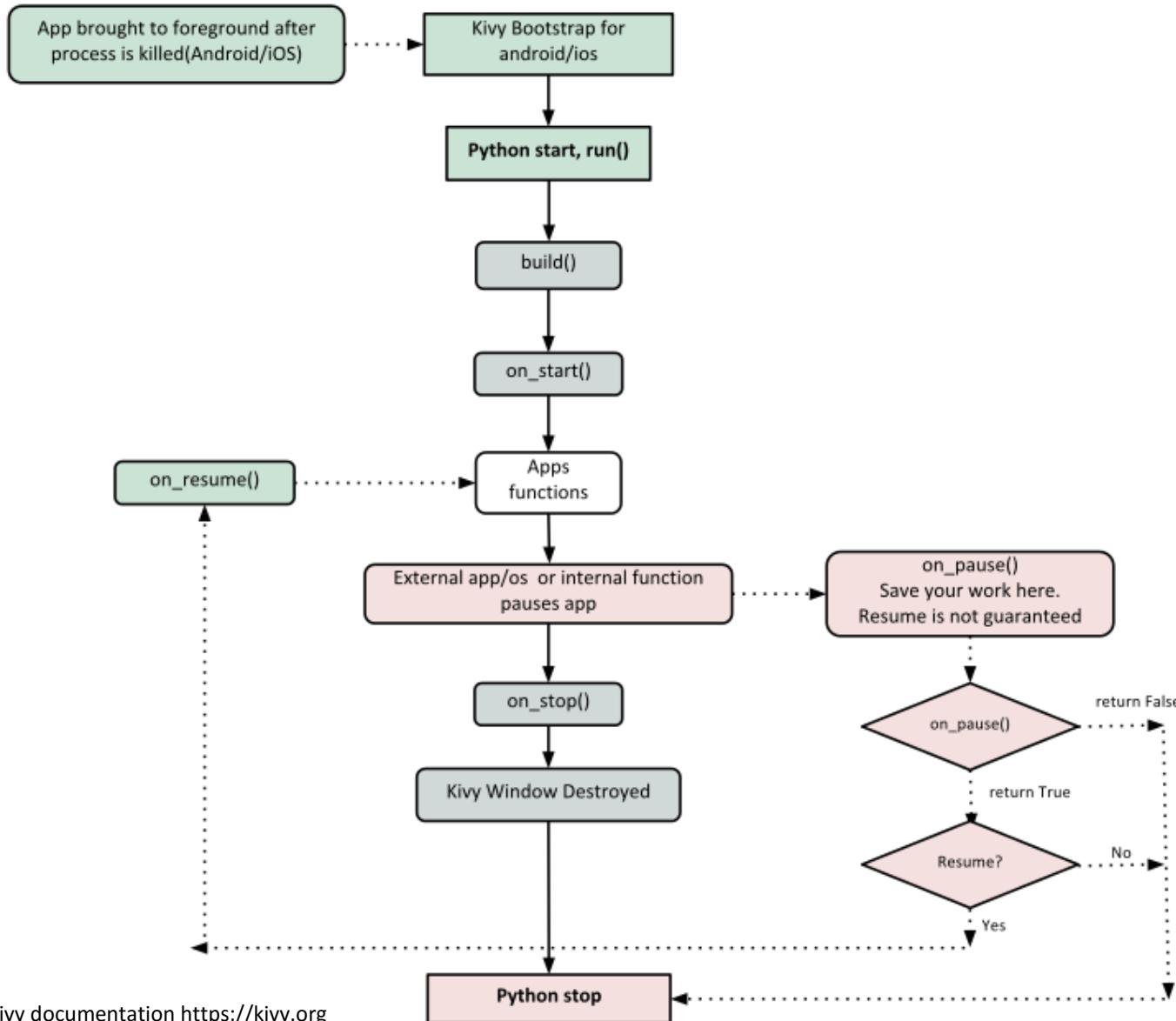


image source: Kivy documentation <https://kivy.org>

Widgets



```
class LoginScreen(GridLayout):
    def __init__(self, **kwargs):
        super().__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)
```

See
e02_widgets.py

KV language



- Separation of presentation and logic
- Loading
 - Implicit
MyApp -> my.kv
 - Explicit
`Builder.load_file('path/to/file.kv')`
- kv file content ~ set of rules
 - (instance) rule
Widget:
 - class rule
<Widget>:

KV language



- **kv keywords**
 - **app** – instance of application.
 - **root** – base widget/template in the current rule
 - **self** – current widget
- **syntax**

See
e03_kv_language.py

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

#:set name value
```

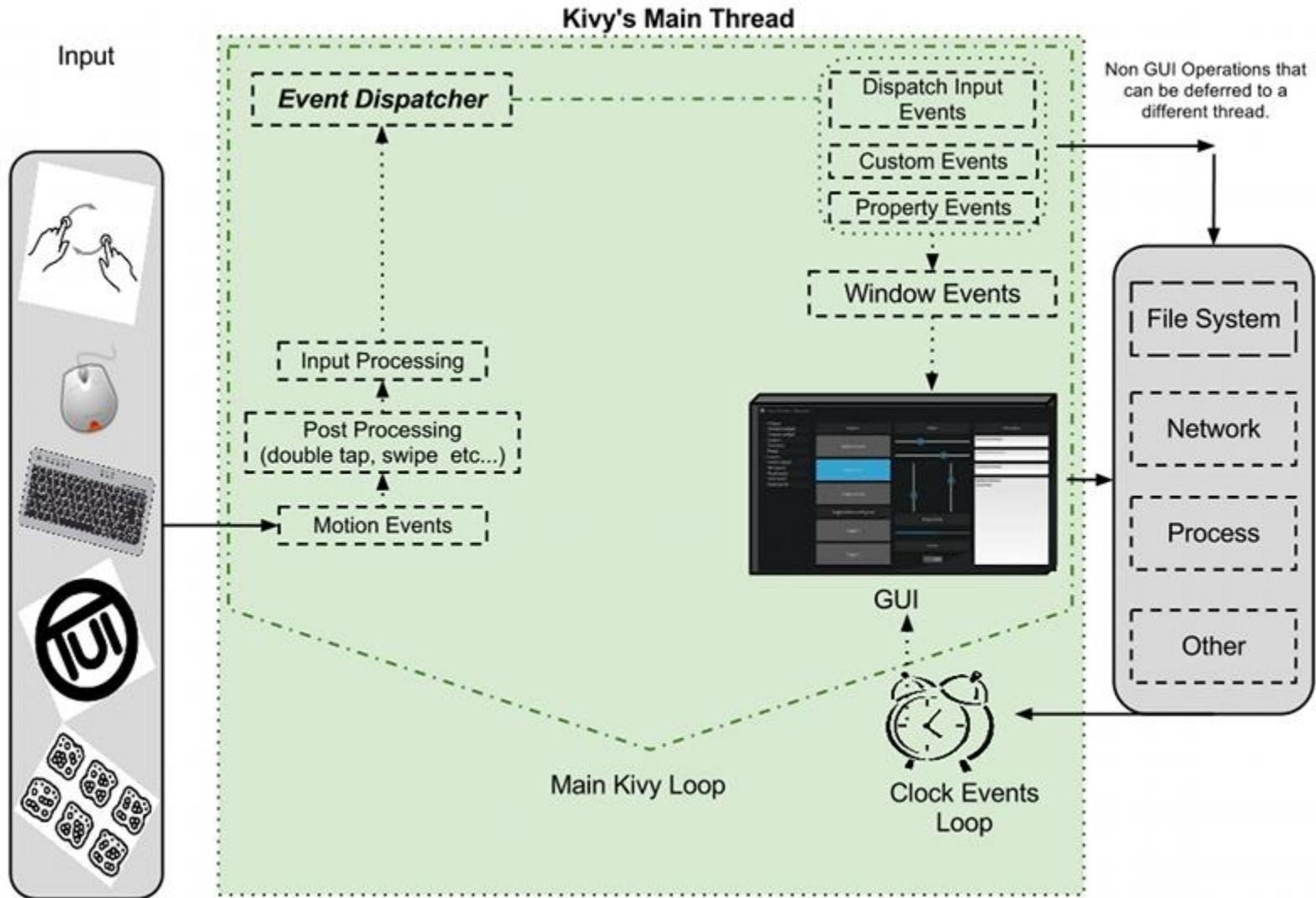
kv

Python
equivalent

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

name = value

Events



Events

- Single thread UI
 - do not block the UI thread
- Repetitive events

```
count = 0

def my_callback(dt):
    global count
    count += 1
    if count == 10:
        print('Last call')
        return False
    print('My callback is called')

Clock.schedule_interval(my_callback, 1 / 30.)
```

Stop repeating

Use
schedule_once(callback, delay)
for non-repetitive events

Events



- Widget events
 - property events – widget changes its position or size, ...
 - widget-defined event – e.g., button pressed or released

Own events

```
class MyEventDispatcher(EventDispatcher):  
    def __init__(self, **kwargs):  
        self.register_event_type('on_test')  
        super().__init__(**kwargs)  
  
    def do_something(self, value):  
        self.dispatch('on_test', value)  
  
    def on_test(self, *args):  
        print "I am dispatched", args  
  
def my_callback(value, *args):  
    print("Hello, I got an event!", args)  
  
ev = MyEventDispatcher()  
ev.bind(on_test=my_callback)  
ev.do_something('test')
```

Has to be implemented by all objects that produce events



Properties

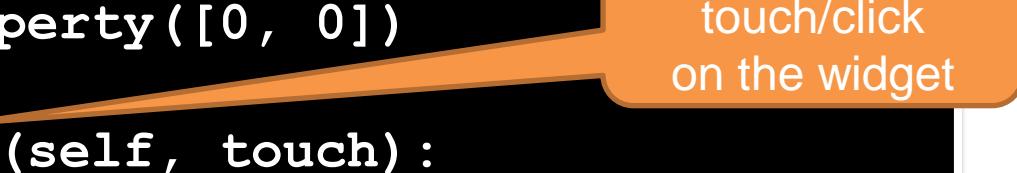
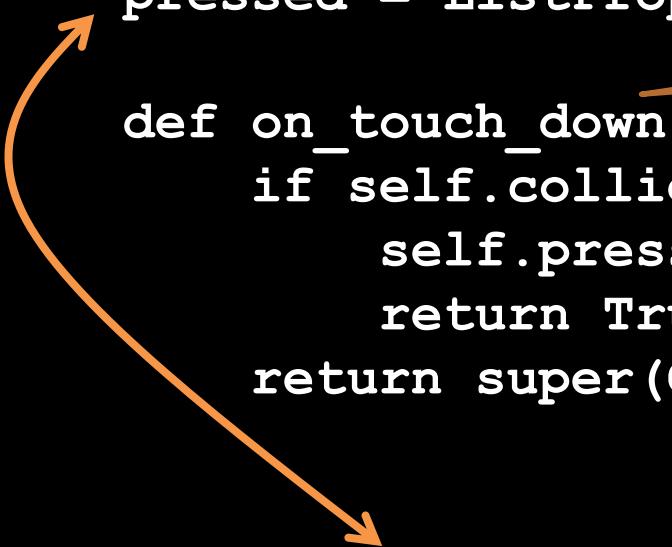


- produce events such that when an attribute of an object changes, all properties that reference that attribute are automatically updated
- **StringProperty**
- **NumericProperty**
- **BoundedNumericProperty**
- **ObjectProperty**
- **DictProperty**
- **ListProperty**
- **OptionProperty**
- **AliasProperty**
- **BooleanProperty**
- **ReferenceListProperty**

Property and events



```
class CustomBtn(Widget):  
    pressed = ListProperty([0, 0])  
  
    def on_touch_down(self, touch):  
        if self.collide_point(*touch.pos):  
            self.pressed = touch.pos  
        return True  
    return super(CustomBtn,  
                self).on_touch_down(touch)  
  
def on_pressed(self, instance, pos):  
    print(f'pressed at {pos}')
```



properties, by default, provide an on_<property_name> event

Property and events



- Binding to properties
 - to monitor changes

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

See
e04_properties.py

Input



- Support for multiple different inputs
 - mouse, touchscreen, accelerometer, gyroscope,...
- MotionEvent
 - Touch events – contain at least position (x,y)
 - No touch events – all other events
- Methods
 - on_motion()
 - on_touch_down()
 - on_touch_move()
 - on_touch_up()

Motion event profiles



- Depends on HW
- Which info is available

```
def on_touch_move(self, touch):  
    print(touch.profile)  
    return super().on_touch_move(touch)
```

See
e05_event_profiles.py

Motion event profiles



- Profiles

- angle 2D angle. Accessed via the a property
- button Mouse button ('left', 'right', 'middle', 'scrollup' or 'scrolldown'). Accessed via the button property
- markerid Marker or Fiducial ID. Accessed via the fid property
- pos 2D position. Accessed via the x, y or pos properties
- pos3d 3D position. Accessed via the x, y or z properties
- pressure Pressure of the contact. Accessed via the pressure property
- shape Contact shape. Accessed via the shape property

Touch event



- Special type of motion event
 - has `is_touch` property True
- Dispatched to **ALL** currently displayed widgets

```
def on_touch_down(self, touch):  
    if self.collide_point(*touch.pos):  
        pass
```

See
`e06_touch_event.py`

Taps



- Double & triple taps

```
def on_touch_down(self, touch):  
    if touch.is_double_tap:  
        ...
```

```
def on_touch_down(self, touch):  
    if touch.is_triple_tap:  
        ...
```

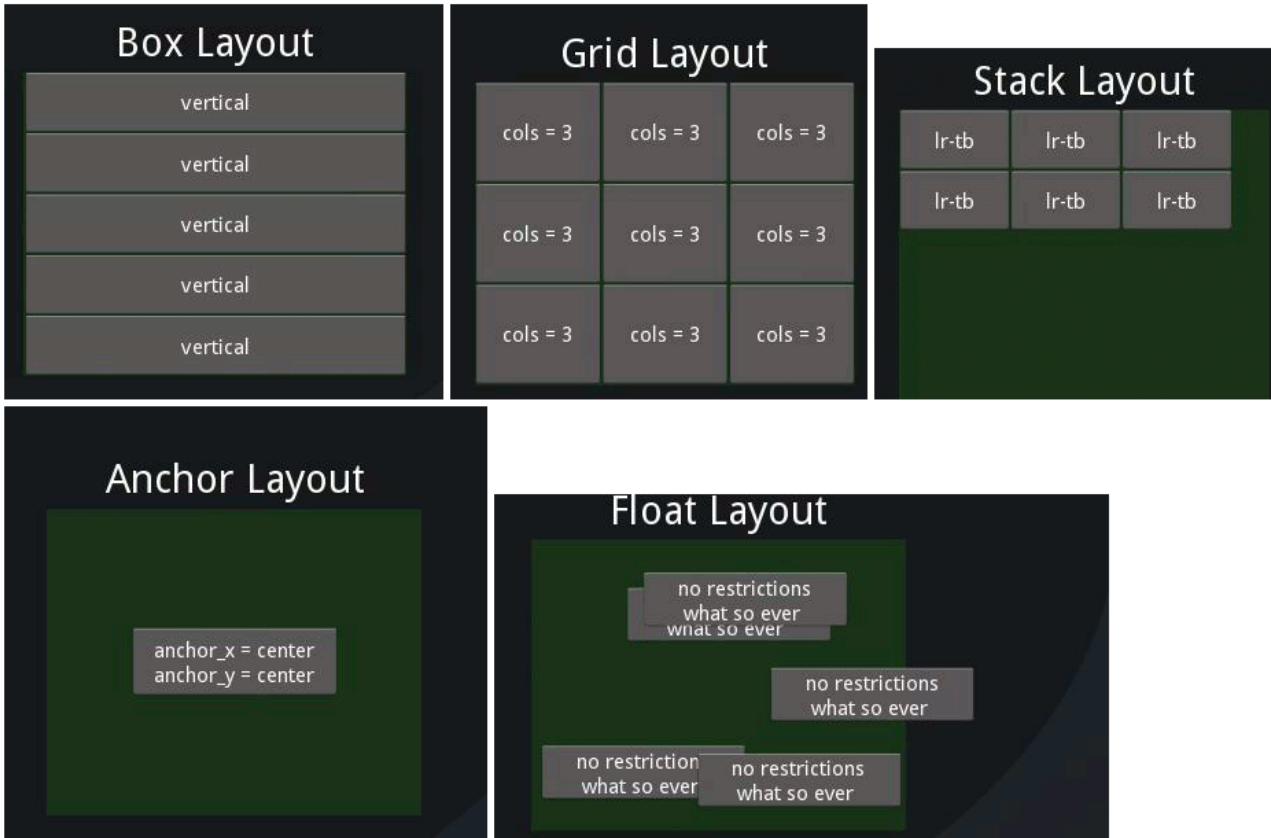
See
e07_double_triple_tap.py

Widgets



- Organized in the widget tree
 - all children in the children attribute
- Methods to manipulate widgets
 - add_widget()
 - remove_widget()
 - clear_widgets()
- Z index
 - lower indexed widgets will be drawn above those with a higher index
 - `root.add_widget(widget, index)`
 - default is 0 – widgets added later are drawn on top of the others

Layouts



- plus Relative layout
 - like Float, but children positions are relative to layout position, not the screen

Layouts



- **size_hint**
 - values from 0 to 1 or None
 - **defaults** (1, 1)
 - if the widget is in a layout, the layout will allocate it as much place as possible in both directions (relative to the layouts size)
- **pos_hint**
 - a dict
 - **defaults** to empty
 - attributes – x, y, right, top, center_x, center_y
 - relative to its parent
- **layouts honor both differently**
 - see documentation

Canvas



- Each widget has a canvas
 - graphical representation of the widget
- Canvas – drawing board with drawing instructions
 - context instructions
 - vertex instructions
- “three” canvases
 - canvas – “main” canvas
 - canvas.before – drawn before the main
 - canvas.after – drawn after the main
- (0,0) is BOTTOM, left
- Instructions are “permanent”
 - can be updated

See
`e08_canvas.py`
`e09_canvas.py`

Popups



- Dialog windows

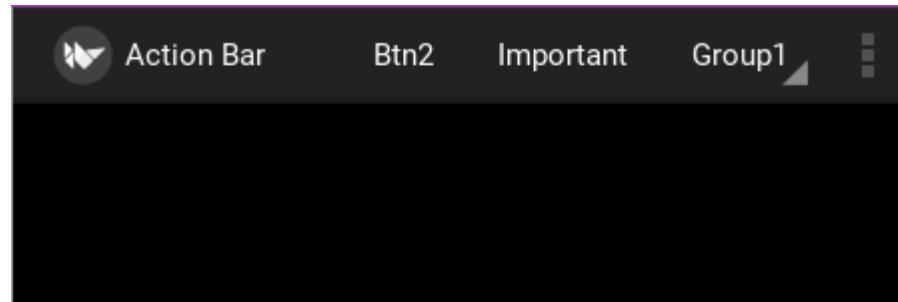
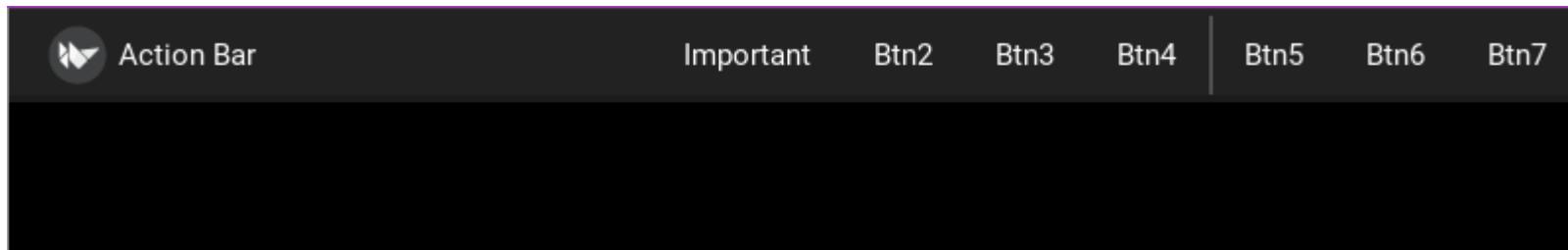
```
popup = Popup(title='Test popup',  
             content=Label(text='Hello world'),  
             size_hint=(None, None), size=(400, 400))  
popup.open()
```

- by default auto-dismissible
 - tap outside the popup
 - auto_dismiss=False
- close from code
 - popup.close()
 - bindable

See
e10_popup.py

ActionBar

- “menu”
 - ActionView
 - ActionButton
 - ActionGroup
 - ActionOverflow



See
e11_bar.py

Widgets IDs



- Referencing to kv file

```
<MyWidget>:  
    Label:  
        id: my_label
```

```
class MyWidget(Widget):  
    def on_button_press(self):  
        self.ids.my_label.text = 'pressed'
```

- Or

```
<MyWidget>:  
    my_label: my_label  
    Label:  
        id: my_label
```

```
class MyWidget(Widget):  
    my_label = ObjectProperty(None)
```

See
e12_ids.py

Kivy on Android



- Deployment
 - Buildozer
 - python-for-android
 - Kivy Launcher
- Pylar
 - pythonic, platform-independent API to use features commonly found on mobile platforms
- Pyjnius
 - access Java classes directly from Python

See
kivy_launcher_apps

See
e13_pylar.py

Task

- Create a calculator

