Kivy

Python everywhere

~~Our Journey to Kivy?~~

Nah, boring. Let's focus on something way cooler.

Kivy: an aerial view

Kivy: an Aerial View

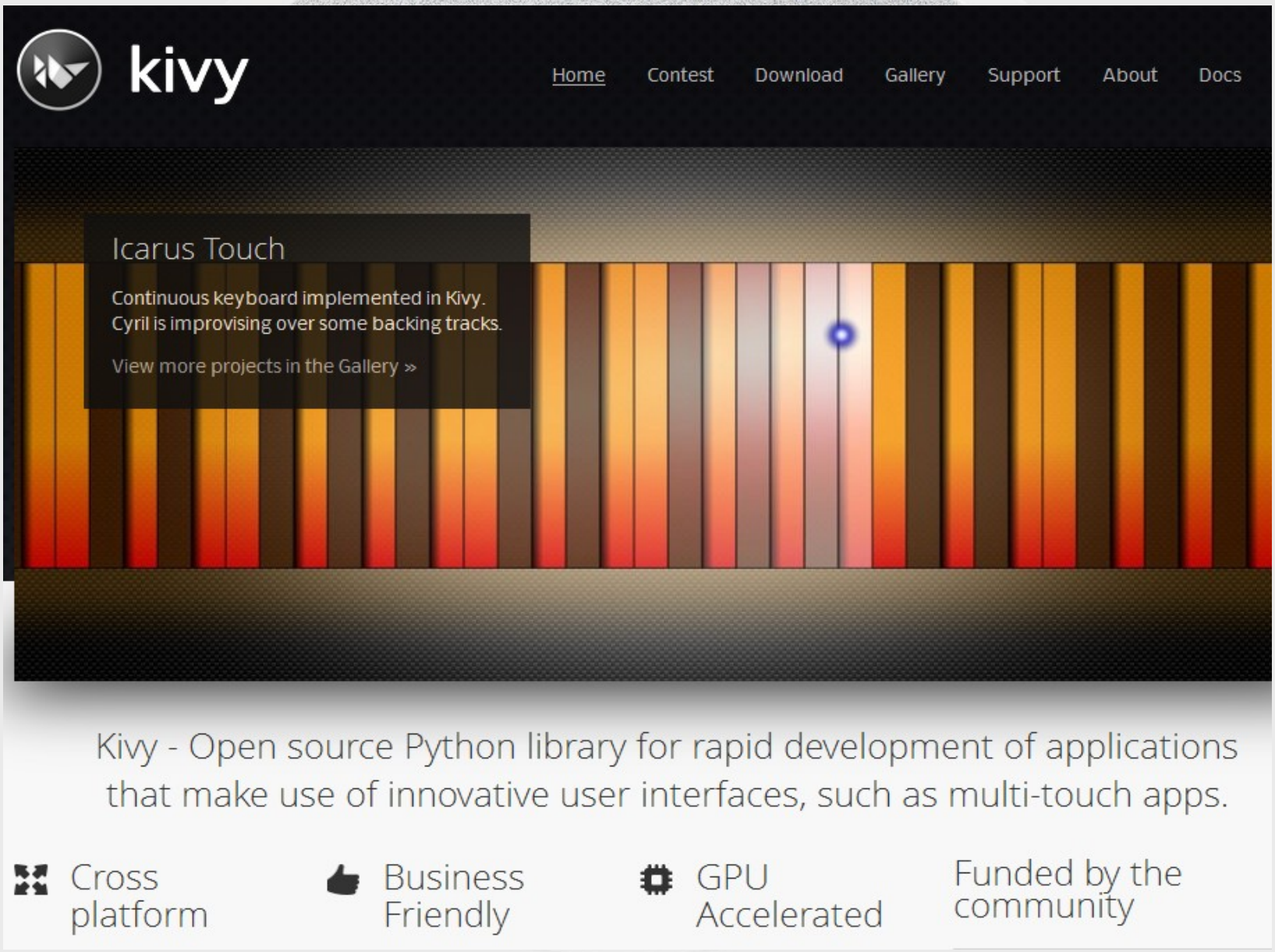
Let's do a quick fly over of the framework.

Yes, I will throw too much information at you.

That is intentional: we want an idea of the scope on Kivy, not it's details

Because Kivy is special. Very special.....

What is Kivy?


kivy
Home
Contest
Download
Gallery
Support
About
Docs
Icarus Touch
Continuous keyboard implemented in Kivy.
Cyril is improvising over some backing tracks.
View more projects in the Gallery »
Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.
Cross platform
Business Friendly
GPU Accelerated
Funded by the community

Why choose Kivy? Official reasons

It's Fresh

- Unqiue and pretty, transitions and animations make it fluid, responsive and interactive.
  Note:
  Kivy does not adopt your native OS look and feel. It gives you your own, consistent UI, everywhere.

Fast

- Uses Cython, so your code is compiled to native C.
- Extensive and highly optimized use of the GPU.

Flexible

- OOP and event driven
- Uses and provides proven design pattens:
  Observer Pattenr
  Single Responsivbility Principle
  Separation of Converns
  Separation of login and UI

Why choose Kivy? The real reasons

It's Awesome!

Kivy is much more that just a complete widgets toolkit. It provides:

- a complete Python standard library (pyython-for-android, python-for-ios)
- a rich set of libraries (Factory, Clock, Atlas, Logger etc)
- OS and device abstraction (Metrics, InputProvers, Core providers)
- a powerful property/event dispatching mechanism
- a set of modules for assisting in developing and debugging
- extensibility though kivy exentions and the Kivy garden
- it's stable, proven, well designed and just damn sexy

Why choose Kivy? More reasons

Kivy provides thoughful, easy-to-use functionality to allows costly IO to be done in background threads(using callbacks).

This makes it easy to ensure your UI stays responsive without worrying about threading.

- `UrlRequest(url, on_success=None, on_redirect=None, ...)`
- `AsyncImage().source = url`

It includes standard Animation classes handled without blocking your main thread.

```
anim = Animation(x=50, y=200, duration=2)
anim.bind(on_complete=callback)
anim.start(widget)
```

Firstly, lets go into the details of what makes kivy different.

To do that, it's best to first understand how the core design choices have shaped Kivy's architecture.

Difference 1: Optimized for the GPU

Drawing is done by OpenGL, and OpenGL is NOT a canvas!

Your Python runs on the CPU, but OpenGL runs on the GPU.

OpenGL is a collections of instruction groups, not an image.

When your widget properties change, the instruction sets do not.

Updating these instructions sets whenever your Python obect changes? Horribly inefficient.

Solution?

Observe!

Difference 1: The gotcha

Why does my widget not move?

Because you are changing it's Python properties, not it's OpenGL drawing instructions.

First solution: bind manually in Python.

Second solution: define your UI using the KV language.

Solutions are easy, the trick is knowing why.

Difference 1: The solutions

You can bind and listen for these changes manually in Python.

```
widget.bind(on_pos=callback, on_size=callback)
```

But that quickly becomes clumsy. Kivy offers the KV lang as a better solution.

Not another language?

Yes, but it's a good solution and a best practise for many reasons:

- performs bindings automagically for you.
- separates UI from code/logic.
- gives an clear picture of your widget heirarchy.
- once learnt, it's much faster to compose a UI.

Examples later, but first, another reason Kivy is different.

Difference 2: Built for multi-touch

Kivy is PyMT, reborn, optimized and implemented in OpenGL ES, the subset of Open GL available on mobile devices. It brings with a rich pedigree of multi-touch experience.

Difference 2: The gotcha

But widget A is inside widget B! Why on earth is it in the bottom left corner?

Firstly, bottom left is (0,0), not top left.

Secondly, the widgets heirarchy is NOT a visaul container heirarchy: it's an event bubbling heirarchy.

Containers are used for layout, but widgets are not artificially constrained to areas.

This allows for total control of layout and gives you full control of the entire screen.

It also facilitates seamless layering. Just pop in another layout, and you can seemlessly float anything anywhere.

Difference 2: Another gotcha

What? Wherever I click, all the widgets "on_touch_down" events fire?

Don't think windows and rectangles! That 's very 95.

The idea of rectangular widget with pre-defined borders breaks down in a touch enviroment.

Example.

> You want a widget to receive a swipe in event.
>
> By definition, that event begins outside of the widget.

In a touch envronment, widgets need new degrees of freedom to monitor input.

Kivy widgets are not limited by their size and position in either output or input. These are just properties!

Yes please! Give me the power!

So how do widgets get touch events? Observe!

Difference 2: The solution

Apply the 'Observer Pattern'

By binding to specific events, any widget can react to almost any event with minimal overhead.

This does not prescribe how events are handled.

It allows total flexibility and puts you in control.

It helps write better code in may ways.

They are the perfect examples of solving a problem well, and the solution being applicable across concerns.

Let's get dirty

Almost all of the challenges Kivy faces are solved in an elegant and consistent way: observing.

Ironically, the solutions actually improve the framework and increase it's flexibility.

And it only gets more impressive as you get deeper.

But enough bluster.

Let's get dirty! Let's dive into some details;-)

Python Properties using Decorators

```
5  class DecProperties(object):
6      _text = ''
7
8      @property
9      def text(self):
10         '''The property "getter"'''
11         return self._text
12
13     @text.setter
14     def text(self, value):
15         '''The property "setter"'''
16         self._text = value
```

Okay, so a hidden variable.

Err...@text.setter? Pretty cryptic.

And two "text" methods?

Isn't there a better way?

Python Properties using property()

```
6  class PropProperties(object):
7      _text = ''
8
9      def _get_text(self):
10         '''The "getter"'''
11         return self._text
12
13     def _set_text(self, value):
14         '''The "setter"'''
15         self._text = value
16
17     text = property(_get_text, _set_text)
```

Gee, 4 things.

Okay, still a hidden variable.

Gee, a "getter", a "setter" and then something to combine them into a property?

Please, isn't there a better way?

Python Properties using Kivy

```
7  class KivyProperties(EventDispatcher):
8      text = StringProperty()
```

Yes, welcome to the Kivy awesome!

It looks like a "static" class method, but Kivy magically bind values to each instance.

Want to watch for changes?

```
10     # To listen for changes
11     def on_text(self, instance, value):
12         print "Changing to ", value
```

But wait, it gets better.

Python Properties with Restrictions

Using normal properties, restricting properties to certain values involves something like this:

```
class MyClass(object):
    def __init__(self, a=1):
        super(MyClass, self).__init__()
        self._a = 0
        self.a_min = 0
        self.a_max = 100
        self.a = a

    def _get_a(self):
        return self._a
    def _set_a(self, value):
        if value < self.a_min or value > self.a_max:
            raise ValueError('a out of bounds')
        self._a = a
    a = property(_get_a, _set_a)
```

Using Kivy properties:

```
class MyClass(EventDispatcher):
    a = BoundedNumericProperty(1, min=0, max=100)
```

There are many more: StringProperty, ObjectProperty etc. And yes, you can create our own.

The really awesome thing is they do much more than normal properties...

Kivy Properties are Event Dispatchers

Kivy properties are also subscribable event dispatchers.

```
class MyClass(EventDispatcher):
    a = NumericProperty(1)

def callback(instance, value):
    print('My callback is call from', instance)
    print('and the a value changed to', value)

ins = MyClass()
ins.bind(a=callback)

# At this point, any change to the a property will call your callback.
ins.a = 5    # callback called
ins.a = 5    # callback not called, because the value didnt change
ins.a = -1   # callback called
```

So Kivy properties provide not only a more elegant and concise property syntax, but add the ability to listen for events.

Using normal properties to create an event-driven architecture? Yes please!

They help in many ways: optimization, separation of concerns, assigning single responsibilities, avoiding state variables and object references and well as flexilbilty.

In short, they totally rock!

Touch made easy

Kivy's rich pedigree of experience shows.

Touch = Mouse. Touch is just richer i.e. has some extra properties.

The Touch object is a complete representation of the entire touch event, and in passed into all of these methods:

```
on_touch_down

on_touch_move

on_touch_up
```

The 'on_touch-down' event is dispatched to every widget that requests to listen.

If the widget wishes to monitor the event, it can "grab" it in the "on_touch_down" and will then receive all subsequent "on_touch_move" and "pn_touch_up" events".

The Touch Object

The touch object is a specialized instance of the MotionEvent class.

These are ojects generated by InputProviders. Nicely abstracted and easy to use.

Here are some of the Touch object's interesting propeties:

```
'apply_transform_2d', 'clear_graphics', 'copy_to', 'depack', 'device',
'distance', 'double_tap_time', 'dpos', 'dsx', 'dsy', 'dsz', 'dx', 'dy',
'dz', 'grab', 'grab_current', 'grab_exclusive_class', 'grab_list',
'grab_state', 'id', 'is_double_tap', 'is_mouse_scrolling', 'is_touch',
'is_triple_tap', 'move', 'opos', 'osx', 'osy', 'osz', 'ox', 'oy', 'oz',
'pop', 'pos', 'ppos', 'profile', 'psx', 'psy', 'psz', 'push',
'push_attrs', 'push_attrs_stack', 'px', 'py', 'pz', 'scale_for_screen',
'shape', 'spos', 'sx', 'sy', 'sz', 'time_end', 'time_start',
'time_update', 'triple_tap_time', 'ud', 'uid', 'ungrab',
'update_graphics', 'update_time_end', 'x', 'y', 'z']
```

As you can see, the Touch object maintains detailed information about the entire interaction so we dont' need to.

Special mention: notice the Z axis? Kivy also supports 3D gesture input, currently via the Leap Motion and Microsoft Kinect.

Grabbing a touch

You can grab a touch if you absolutly want to receive these events:

```
on_touch_move()

on_touch_up()
```

Touch events normally bubble through the widget hierarchy until one of those methods returns True, indicating the event has been handled.

Grabbing a touch ensures you get all it's subsequent events.

Very powerful: it allows widgets to receive touch events independent of visual placement.

Let's look at an example.

Grabbing a Touch: an example

```
1  def on_touch_down(self, touch):
2      if self.collide_point(*touch.pos):
3          # if the touch collides with our widget, let's grab it
4          touch.grab(self)
5          return True  # Indicate we have handled the touch
6
7
8  def on_touch_move(self, touch):
9      # This event can also be used to monitor grabbed touch
10     pass
11
12
13 def on_touch_up(self, touch):
14     # Check if it's a grabbed touch event
15     if touch.grab_current is self:
16         # ok, the current touch is dispatched for us.
17         print('I have been touched!')
18         touch.ungrab(self)  # ungrab or you might have side effects
19         return True  # indicate we have handled the event
20
```

From Touch to Gestures

Touch is great, but how do we use it for detecting gestures?

Converting input points to gestures is, err, challenging i.e. damn difficult.

But don't worry, Kivy does it all for you.

Let's look at an example.

Gesture Recognition

```
3  # Create a gesture
4  g = Gesture()
5  g.add_stroke(point_list=[(1, 1), (3, 4), (2, 1)])
6  g.normalize()
7  g.name = "triangle"
8
9  # Add it to database
10 gdb = GestureDatabase()
11 gdb.add_gesture(g)
12
13 # And for the next gesture, try to find a match!
14 g2 = Gesture()
15 g2.add_stroke(point_list=[(1, 1), (3, 4), (2, 1)])
16 g2.normalize()
17 print gdb.find(g2).name  # will print "triangle"
18
19
```

Easy to access Gesture databases built in, with normalization.!

Recording Gestures

```
def simplegesture(name, point_list):
    '''A simple helper function to create and return a gesture'''
    g = Gesture()
    g.add_stroke(point_list)
    g.normalize()
    g.name = name
    return g

class GestureBoard(FloatLayout):
    def on_touch_down(self, touch):
        '''Collect points in touch.ud'''
        with self.canvas:
            touch.ud['line'] = Line(points=(touch.x, touch.y))
        return True

    def on_touch_move(self, touch):
        '''store points of the touch movement'''
        try:
            touch.ud['line'].points += [touch.x, touch.y]
            return True
        except (KeyError) as e:
            pass

    def on_touch_up(self, touch):
        '''The touch is over. Create a gesture and store it'''
        g = simplegesture('cross',
            list(zip(touch.ud['line'].points[::2], touch.ud['line'].points[1::2])))
        gdb = GestureDatabase()
        gdb.add_gesture(g)
```

This means you can record gestures and store them for later comparison against input.

Makes creating a gesture vocabulary dead simple.

The Widgets

We will launch the Kivy Showcase demo from the Kivy Windows portable package.

Self-contained: can be run on systems irrespective on their current Python installation.

Contains: all Kivy source code, examples and a complete Cython install with all Kivy libraries.

About 75mb zipped, but your installation can be much smaller if packaged with PyInstaller.

Other Notable Widgets

The ScreenManager

By default, apps are one 'Screen'.

The ScreenManager allows you to create multiple screens and switch between them.

You get a whole lot of beautiful, fluid transitions for free:

SlideTransition - slide screen in/out, from any direction

SwapTransition - implementation of the iOS swap transition

FadeTransition - shader to fade in/out the screens

WipeTransition - shader to wipe from right to left the screens

Other Notable Widgets

The Scatter

The Scatter is used to build interactive widgets that can be translated, rotated and scaled with two or more fingerss on a multitouch system.

It has as its own matrix transformations which makes possible to perform the rotation /

scaling / translation over the entire child tree without changing their properties.

```
1  from kivy.uix.scatter import Scatter
2  from kivy.uic.image import Image
3
4  # Allow translation on the x-axis only as all properties are enabled by default
5  scatter = Scatter(do_rotation=False, do_scale=False, do_translation_y=False)
6  image = Image(source='sun.jpg')
7  scatter.add_widget(image)
8
```

Other Notable Widgets

The CodeInput

Provides a box of editable highlited text.

It supports all the features supported by the TextInput.

Proivides Code highliting for languages supported by pygments along with KivyLexer for KV Language highliting.

```
if __name__ == '__main__':
    from kivy.app import App
    from kivy.uix.boxlayout import BoxLayout

    class TextInputApp(App):

        def build(self):
            root = BoxLayout(orientation='vertical')
            textinput = TextInput(multiline=True)
            textinput.text = __doc__
            root.add_widget(textinput)
            textinput2 = TextInput(text='monoline textinpu
                                   size_hint=(1, None), height=30)
            root.add_widget(textinput2)
            return root

    TextInputApp().run()
```

```
BoxLayout:
    # Double as a Tabbed Panel Demo!
    TabbedPanel:
        tab_pos: "top_right"
        default_tab_text: "List View"
        default_tab_content: list_view_tab

        TabbedPanelHeader:
            text: 'Icon View'
            content: icon_view_tab

        FileChooserListView:
            id: list_view_tab

        FileChooserIconView:
            id: icon_view_tab
            show_hidden: True
```

Other Notable Widgets

The Popup

Creates modal popups.

Uses lighbox effect to grey our background window.

Very flexible: you pass it a 'content' property which can contain any widgets or containers.

```
from kivy.uix.popup import Popup
from kivy.uix.label import Label

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

Other Notable Widgets

Layouts

Layouts - the proper way to handle UI.

Provide very fluid and elegant mechanisms to scale UI for different screen resolutions and densities.

Each works differently, but generally use 'pos_hint' and 'size_hint' to request placement.

Customizing Widgets

Many options.

- create you own widget inheriting the EventDispatcher.
- subclass and alter (widget.background_down, widget.background_up etc).
- replace the kivy atlas with your own atlas image.

The Kivy Garden

Not every widget or add-on you want can or should be included with Kivy.

The Garden provides a repository of additional widgets that anyone can contribute to.

It's maintained by the community, not the Kivy core team.

Kivy ships with a garden tool and it's trivially easy to use: .

```
# Installing a garden package
garden install graph

# Upgrade a garden package
garden install --upgrade graph

# Uninstall a garden package
garden uninstall graph

# List all the garden packages installed
garden list

# Search new packages
garden search

# Search all the packages that contain "graph"
garden search graph
```

The Widgets in the Garden

Currently, widgets in the Garden include graphs, custom spinners, navigationdraws, DatePickers etc.

Some notable ones:

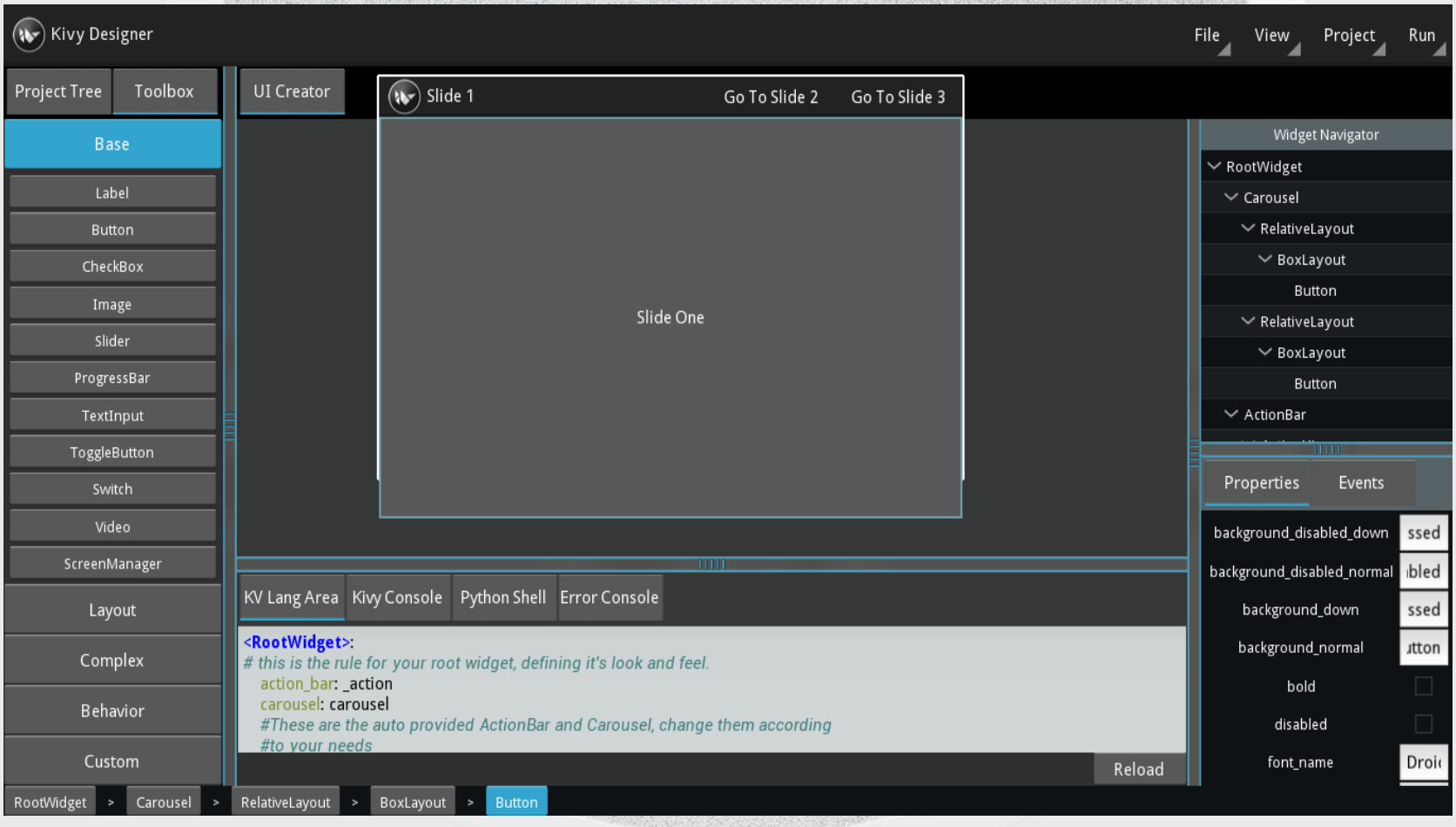
- CEPython – an embedded Chromium browser using CEFPython (not yet stable)
- Roulette – provedes a way of selecting values like iOS and android date pickers.
- Filechooserthumbview - FileChooserThumbView for Kivy.
- Particlesystem – A particle engine for Kivy framework

And many more...

Something else Exciting

The Kivy-Designer – A WYSIWYG UI designer.

Currently being worked on by Abhinav Abhijangda as part of his Google Summer-Of-Code.



Kivy Modules

Kivy provides some amazing tools for development and debugging.

They are easy to use and activate

You can also roll your own (simple module with start(win, ctx) and stop(win, ctx) methods)

They include:

touchring: circles each touch.

monitor: shows a small FPS and activity graph.

keybinding: bind keys to actions, such as a screenshot.

recorder: record and playback events.

screen: emulate different screens.

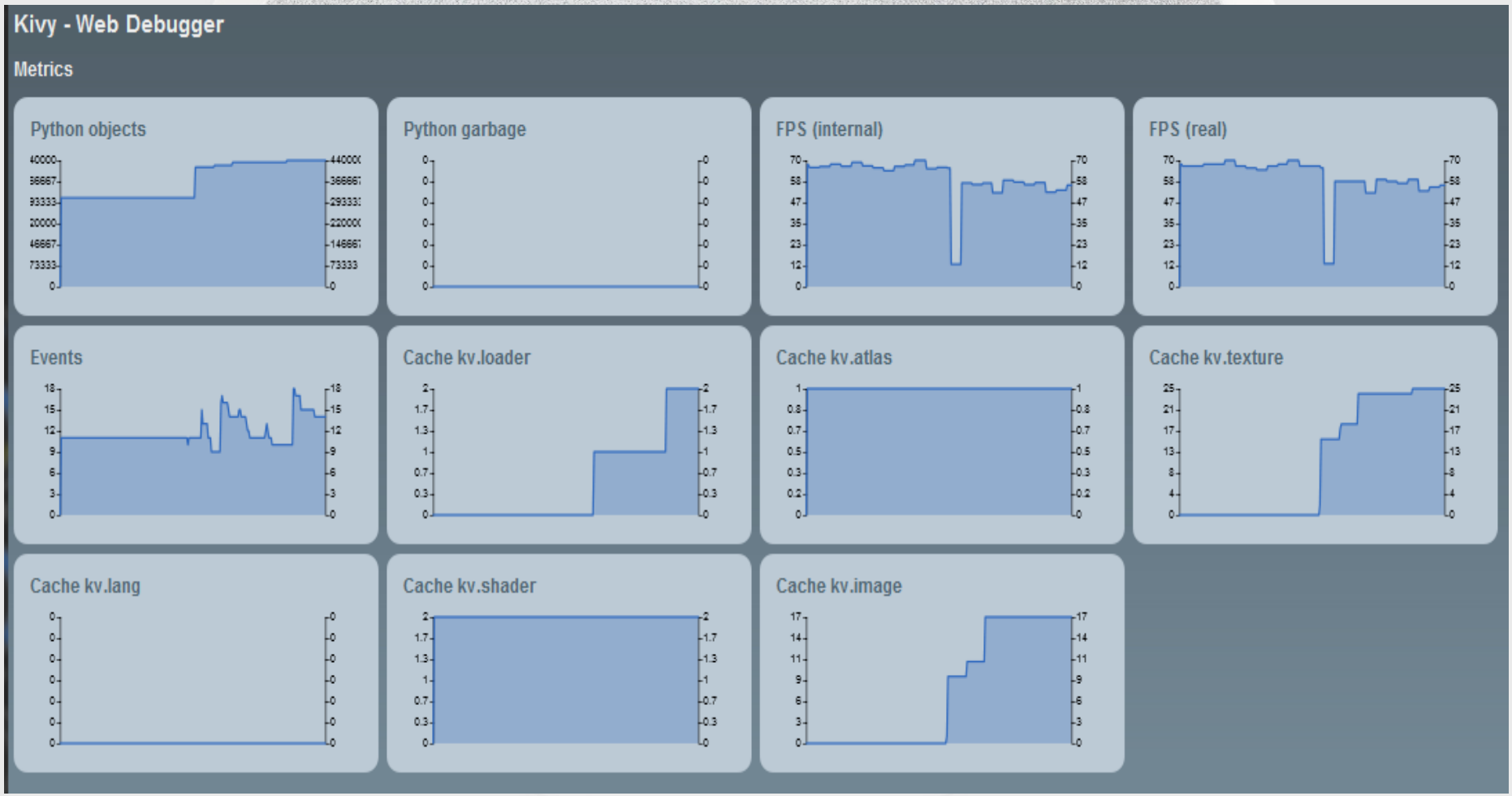
inspector: examine your widget heirarchy

webdebugger: realtime examination of your app internals via a web browser

Webdebugger Module

Provides realtime inspection of your apps internals.

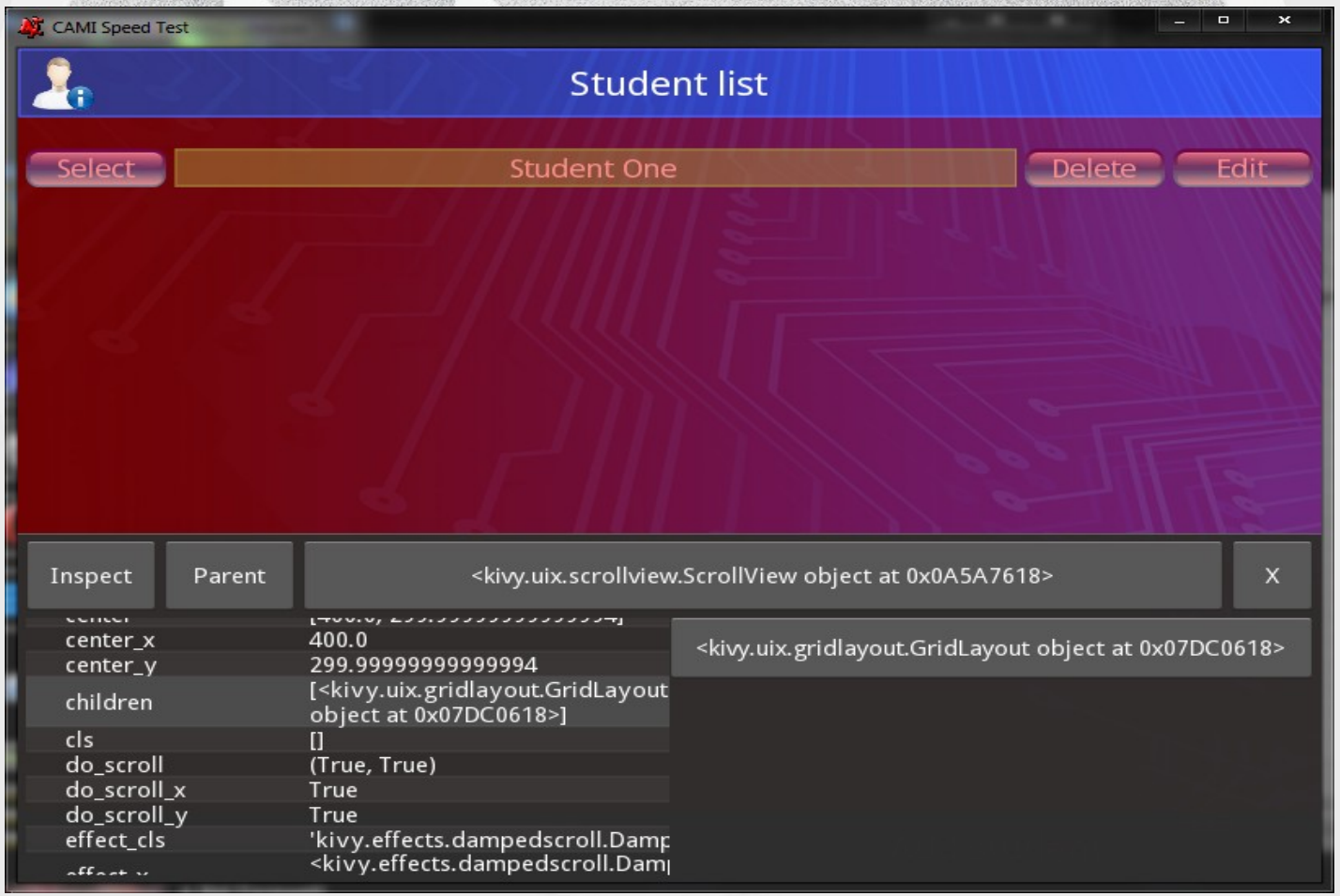
It doest his using an embedded micro-webframework called flask + embedded jQuery.



Inspector Module

The inspector allows for interactively travesing your entire widget heirarchy.

It provides a complete analysis of each widget, including it's children.



Interactive Launcher

* This is not a module, just a cool tool!

The InteractiveLauncher provides a user-friendly python shell interface to an App so that it can be prototyped and debugged interactively. It supports IPython!

```
class TestApp(App):
    def build(self):
        return Widget()


i = InteractiveLauncher(TestApp())
i.run()
i.       # press 'tab' to list attributes of the app
i.root.  # press 'tab' to list attributes of the root widget

# App is boring.  Attach a new widget!
i.root.add_widget(MyPaintWidget())

i.safeIn()
# The application is now blocked.
# Click on the screen several times.
i.safeOut()
# The clicks will show up now

# Erase artwork and start over
i.root.canvas.clear()
```

Libraries

Kivy provides a rich set of libraries offering extra functionality.

- Animations and transitions, Atlas, Clock, Factory, Logger, Metrics, Vector, Adaptors, Effects (ScrollEffect, DampedScrollEffect, OpacityScrollEffect + garden + roll you own), Gesture recording, playback and recognition, Shader and Stencil instructions, Asynchronous Data Loader, Utils, UrlRequest, ReStructuredText and Virtual KeyBoard.

- Core abstractions including:

Audio, Camera, Clipboard, OpenGL, Image, Spelling, Text, Video, Window, InputProviders.

Let's quickly look as some interesting libraries.

Metrics

Metrics provide a seamless, powerful way to work with different screens and resolutions.

100.1 Dimensions

As you design your UI for different screen sizes, you'll need new measurement unit to work with.

Units

pt Points - 1/72 of an inch based on the physical size of the screen. Prefer to use sp instead of pt.

mm Millimeters - Based on the physical size of the screen

cm Centimeters - Based on the physical size of the screen

in Inches - Based on the physical size of the screen

dp Density-independent Pixels - An abstract unit that is based on the physical density of the screen. With a `Metrics.density` of 1, 1dp is equal to 1px. When running on a higher density screen, the number of pixels used to draw 1dp is scaled up by the factor of appropriate screen's dpi, and the inverse for lower dpi. The ratio dp-to-pixes will change with the screen density, but not necessarily in direct proportions. Using dp unit is a simple solution to making the view dimensions in your layout resize properly for different screen densities. In others words, it provides consistency for the real-world size of your UI across different devices.

sp Scale-independent Pixels - This is like the dp unit, but it is also scaled by the user's font size preference. It is recommended to use this unit when specifying font sizes, so they will be adjusted for both the screen density and the user's preference.

Font sizing done right

Font sizes are challenging on almost all Operating Systems and platforms.

Points? So, how does that relate to my widget height? Oh, I must calculate actual width and height using this API or that one....

In Kivy, font_size is the height of the font and can be set using any metric.

It's as simple as that.

Screen Metrics Done Right

Screens feature 3 core properties that determine how things appear.

All of these are trivial to access via the metrics module and trivial to log for later emulation.

```
from kivy.metrics import Metrics
from kivy.logger import Logger

Logger.info("app.py: Screen density = " + str(Metrics.density()))
Logger.info("app.py: Screen dpi = " + str(Metrics.dpi()))
Logger.info("app.py: Screen fontscale = " + str(Metrics.fontscale()))
```

The Factory

The Factory is, predicatably enough, and implementation of the Factory pattern.

Register a class, and then use it anywhere without worrying about scope.

There is another bonus: any classes registered become reconizable and usable from you KV file or string.

```
1  from kivy.factory import Factory
2  Factory.register('Widget', module='kivy.uix.widget')
3  Factory.register('Vector', module='kivy.vector')
4
5  # Then later
6  from kivy.factory import Factory
7  widget = Factory.Widget(pos=(456, 456))
8  vector = Factory.Vector(9, 2)
```

The Utils Library

Kivy provides a few handy, general purpose utilities.

platform() # Returns one of: win, linux, android, macosx, ios, unknown.

intersection(set1, set2) # return intersection of 2 lists

difference(set1, set2)

QueryDict # A dict() that can be queried with dot.

escape_markup(text) # Escape markup characters found in the text.

#Intended to be used when markup text is activated on the Label:

Other Cross-Platform Options

Assuming Kivy was just a widget toolkit (wrong!), what other Python options do we have?

GTK - Killed by the flop of GTK3 + Gnome 3. Breaking compatibility, ignoring community feedback.

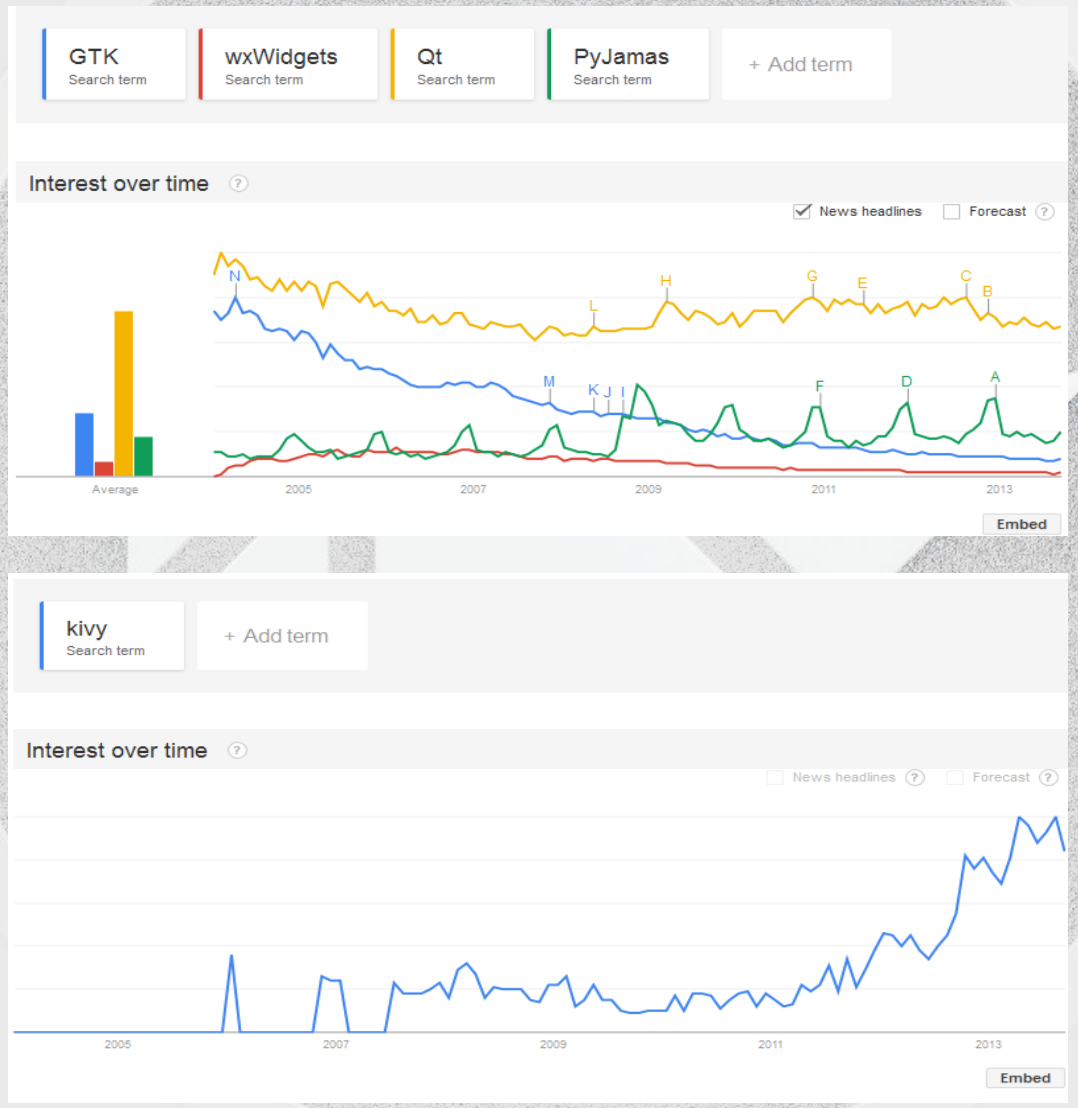
Qt - Limited by licensing concerns, unstable ownership, failure of QML to make significant impact on mobile.

wxWigets - Version compatibility issues, slow development, no Python 3 support.

PyJamas - Complicated, multiple layers of dependencies, slow, recent ownership controvery.

None of them come close to offering the ease-of-development, ease-of-deployment and flexibility Kivy does.

Cross-platform trends



Even Guido loves Kivy!

2012 saw the Python foundation donate $5000 dollars to Kivy to help the Py2-to-3 efforts.

So, why not use Kivy?#1

With all this yummmy goodness, the question should not be "Why Kivy?" but "Why not Kivy?".

Only 3 possible reasons that come to mind.

1. I want it to run in a browser.

   Nope, doomed to Javascript I'm afraid. (Duh!)

So, why not use Kivy?#2

2. I want to leverage iOS/Windows/SharePoint/Android specific features.

You often can do these things in Kivy, but does not make sense because you then depend on the OS anyway.

Involves extra effort, so why use a cross-platform solution? Just use native.

Notes:

Python-for-android inlcudes PyJenius for accessing Java classes.

Working examples of android GPS access and other android specific features exists.

Use of home and back buttons, resume and pause are already supported.

Accessing iOS functions from Kivy-iOS? Sorry, no idea ;-)

So, why not use Kivy?#3

3. We depend heavily of high level document display and printing.

Kivy's support for high level document display is weak. Rst in the only supported one by deafult.

No HTML renderer, but Kivy Berklium and the garden's CEFPython are trying.

No SVG. There might be ways of packaging and SVG renderer such as LibRSVG, but that is desktop only.

No PDF. Yet. Mathieu/tito has got it a version working Linux + Windows, and someone else on MacOSX, but there does not seem to be any current activity on this issue.

Note:

It is trivially easy to request the OS to open these documents with the default viewer. You only have an issue if you need to display them embedded inside of the application.

Otherwise, use Kivy!

Kivy a truly portable, fast, powerful, mature full stack Python framework.

Produces fluid, beautiful, interactive, multi-touch capable apps that run almost everywhere.

Provides libraries for complete OS abstraction.

Has a rich history of multi-touch experience and provides dead-simple ways of using gestures.

Your apps look and behave the same everywhere - no need to cater for OS specific norms.

It's released under MIT license, so it's commerce friendly.

It's hosted on Github and welcomes contributions from the community.

It has a high quality codebase, stricly maintained by black-belt ninja Pythonista's.

It has a growing, vibrant and helful community with very active forums.

As a Developer

I love Kivy. Staight up. But that love has been hard earned.

Once learnt, it's extremely producitve. It's quick and easy to produce exciting applications.

As a well-designed, flexible and stable framework: a pleasure to work with.

It's really motivating for the team: free, on-my-device and sexy.

The Open source license makes me fuzzy inside: to give Kivy to the entire world for free is an act of the highest nobility. Respek!

In Closing

Kivy is an awesome, future-facing framework that deserves more attention.

It's epitomises the promise of Python everywhere, and adds Cython + GPU optimization for speed and fluidity.

Kivy comes from the future and has years of multi-touch experience under it's belt.

But you need to see it to understand how exciting it is.

http://kivy.org

http://kivy.org/#gallery

Huge thanks to all the Kivy devs, especially:

Tito (Mathieu Virbel)

qua-non (Akshay Aurora)

tshirtman (Gabriel Pettier)