PYTHON ACADEMY

DESENVOLVA APPS PARA ANDROID E IOS COM PYTHON E KIVY

Nesse ebook vamos explorar o desenvolvimento de aplicações mobile utilizando Python e Kivy. Aprenda a desenvolver aplicativos para Android, iOS, Desktop e muito mais com Kivy!

PYTHONACADEMY.COM.BR

Você já teve uma ótima ideia de aplicativo revolucionário que ia te deixar bilionário da noite pro dia, mas percebeu que teria que aprender Objective-C, C#, Java **e** (ênfase no **E**) .NET para lançá-lo?

Pois é, em Python temos um *framework* incrível para construir aplicativos nativos que podem ser empacotados para essas diversas plataformas!

Portanto, se você quer desenvolver para Android, iOS, Windows, Linux ou Mac OS utilizando nosso querido Python, o Kivy é uma ótima aposta!

Com isso, adicionamos mais uma usabilidade ao leque de possibilidades do Python: aplicações web (utilizando, por exemplo, Flask ou Django - que eu recomendo **fortemente**)

Com isso, além de podermos usar Python para fazermos aplicações desktop, aplicações distribuídas, aplicações web (utilizando, por exemplo, Flask ou Django - que eu recomendo **fortemente**), o Python também pode ser usado para desenvolver **aplicações multiplataformas**.

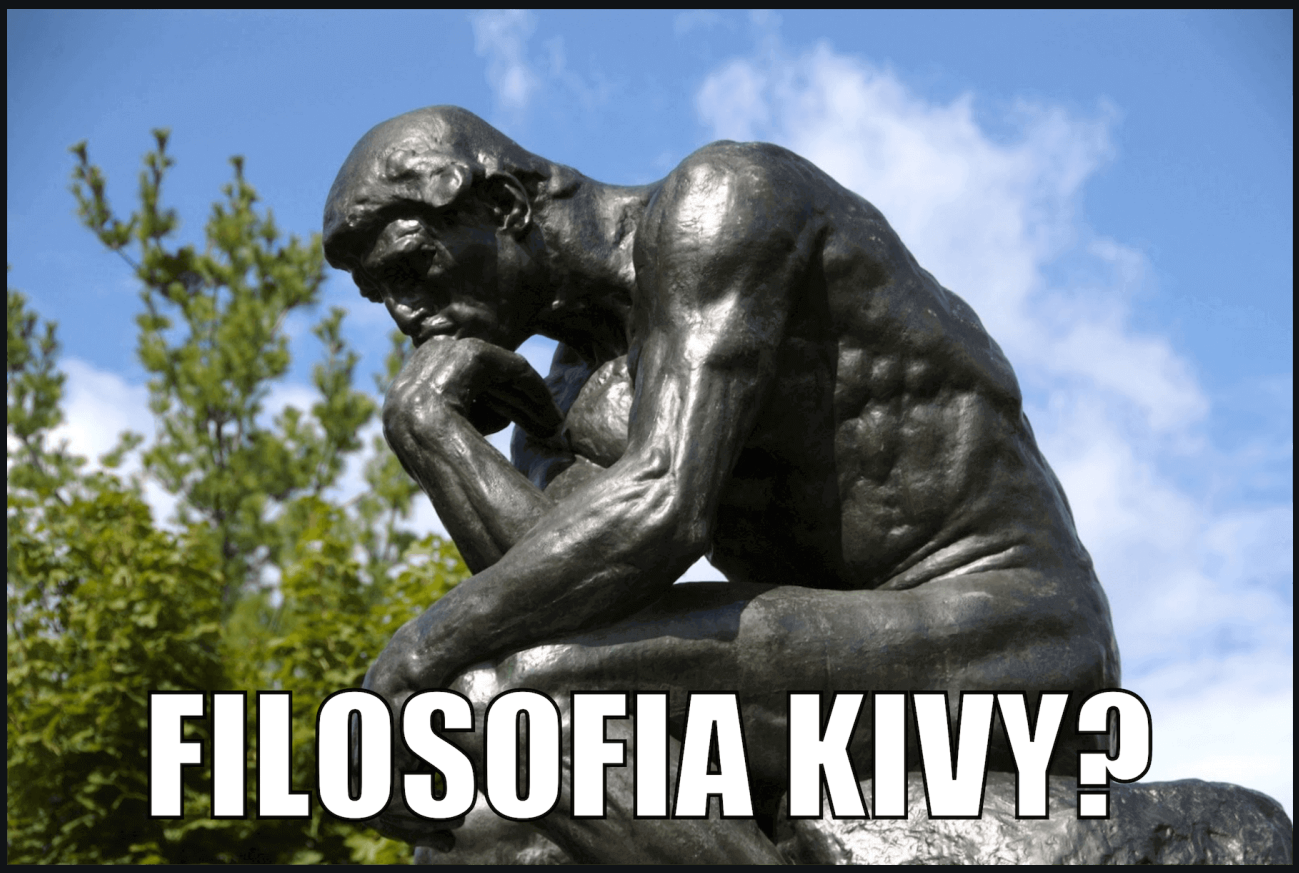
Outras vantagens do Kivy: ele é totalmente grátis, já está em suas versões estáveis e é muito bem documentado.

Com todas essas vantagens, só nos resta cair de cabeça no Kivy!

Nesse post eu pretendo mostrar pra vocês que o Python pode ser usado em seu projetos pessoais, trazendo simplicidade na hora de desenvolver e entregando uma solução completa para diversas plataformas!

Nesse post, vamos: - Aprender o que é o Kivy. - Configurar nosso computador para utilizá-lo. - Explorar algumas de suas funcionalidades. - Entender sua arquitetura.

A Filosofia Kivy



Antes de começarmos a desvendar o Kivy, vou mostrar-lhes os motivos pelos quais devemos utilizá-lo (retirado da Filosofia Kivy).

Moderno

O Kivy foi feito para o presente e o futuro. Por isso, ele foi criado do zero pensando sempre nas interações entre o humano e o computador.

Rápido

Isso se aplica tanto à velocidade de desenvolvimento quanto à velocidade de execução dos aplicativos criados com Kivy.

Para ser rápido, o Kivy foi otimizado de diversas formas. Alguns de seus módulos mais críticos foram feitos inteiramente em C, para utilizar o poder de seus compiladores.

Em outros casos, o Kivy utiliza o poder computacional das GPUs dos dispositivos executando seus códigos.

Flexível

Talvez a melhor característica do Kivy seja seu poder de ser executado em diversas plataformas, tanto mobile quanto desktop.

Ser flexível também significa ser adaptável, isto é, o Kivy se molda rapidamente às novas tecnologias. Muitas vezes, o Kivy se adapta à novos dispositivos, novos protocolos e novos software antes mesmo de serem lançados.

Enxuto

Utilizando o Kivy, podemos escrever aplicações com poucas linhas de código.

Kivy é feito em Python, que é extremamente poderoso, versátil e o melhor, simples!

Baseado nisso, o Kivy possui sua linguagem de descrição, chamada… Kivy! Meio óbvio né?! 😂 Com ela podemos configurar, conectar e organizar os elementos da nossa aplicação de maneira bem simples.

Desenvolvimento e Comunidade

O Kivy é totalmente desenvolvido por profissionais experientes da área e conta com desenvolvedores 100% focados no desenvolvimento do *framework*.

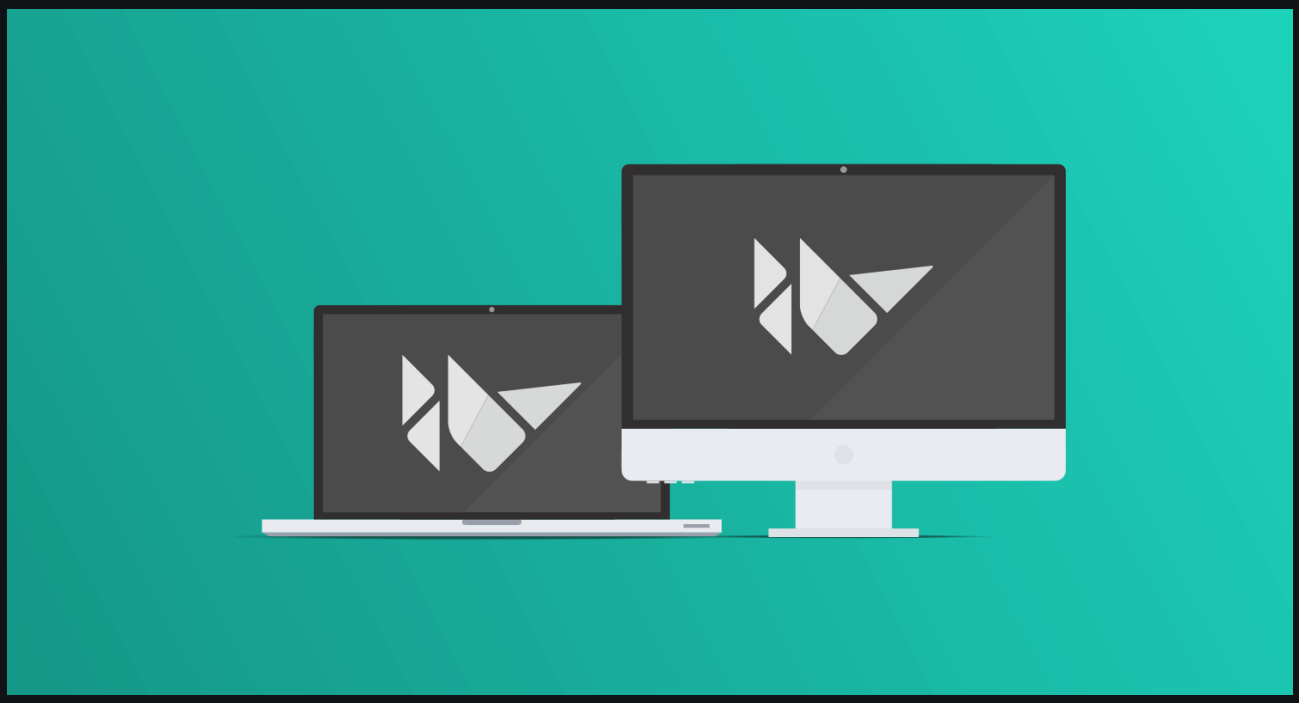
Também possui uma comunidade forte e presente no desenvolvimento de suas soluções.

Grátis

Kivy é totalmente de graça. Mesmo se seu projeto estiver lucrando em cima de soluções desenvolvidas com Kivy, você não deve nada à comunidade Kivy.

Bom, agora que já entendemos do por quê utilizar o Kivy, que tal colocarmos a mão na massa?!

Instalação



Para utilizarmos o Kivy, precisamos primeiro fazer sua instalação. Para isso, acesse a documentação clicando aqui e seguindo o passo a passo referente ao seu Sistema Operacional.

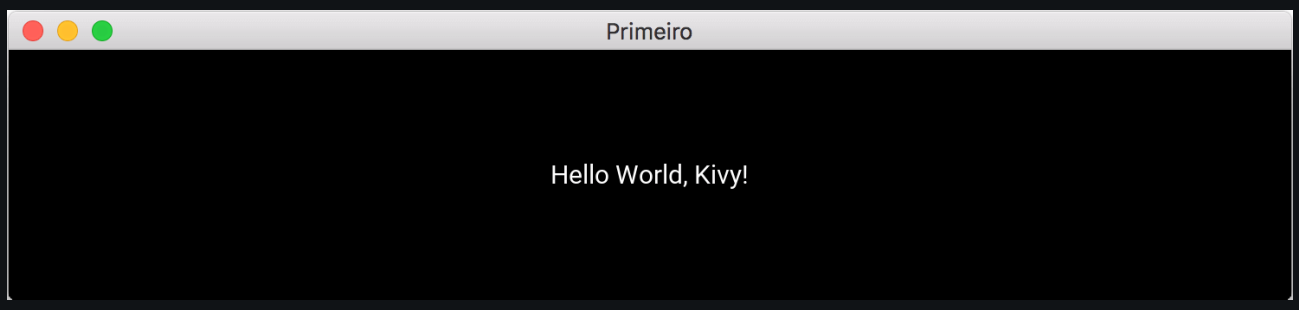
Hello World, Kivy

Agora com tudo instalado e configurado, podemos (até que enfim) executar nossa primeira aplicação utilizando o Kivy.

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

kivy.require('1.9.1')

class PrimeiroApp(App):
    def build(self):
        return Label(text='Hello World, Kivy!')

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

Para isso, coloque o código acima em um script chamado `main.py`, por exemplo, e execute-o com `kivy main.py`.

A saída deve ser algo similar à:



Se algo der errado, não hesite em postar aqui embaixo pois estamos aqui parar ajudar! 👊

Entendendo Nossa Aplicação

Primeiro, toda aplicação deve herdar de `kivy.App` (linha 7). Essa classe controla todo o ciclo de vida da nossa aplicação.

O ponto inicial da nossa aplicação é o método `run()`. Quando essa linha é executada, nossa aplicação entra em execução.

Um ponto importante é saber de onde vem os elementos que compoem nossa aplicação. No Kivy, os elementos estão em `kivy.uix`. Na linha 9, criamos um elemento do tipo `Label` com o valor "Hello World, Kivy!" no atributo `text`.

Dentro desse pacote, temos diversos elementos, como: `Button`, `Image`, `Slider`, `Input`, etc. A lista completa pode ser consultada aqui. Nesse post utilizaremos alguns deles.

Dando continuidade, na linha 8 temos a função `build()` e o que ela faz é construir o elemento raíz da nossa aplicação.

Nesse caso, o elemento raíz é do tipo `Label` com o texto "Hello World, Kivy!".

Com tudo devidamente criado e inicializado, colocamos nossa aplicação em execução com o código:

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

Viram como é simples! Com poucas linhas de código já temos algo sendo mostrado na tela! 😄 E esse é apenas o começo!

Adicionando Elementos

Vamos extender um pouco nossa aplicação adicionando elementos de *input* e colocá-los dentro de um `Layout`.

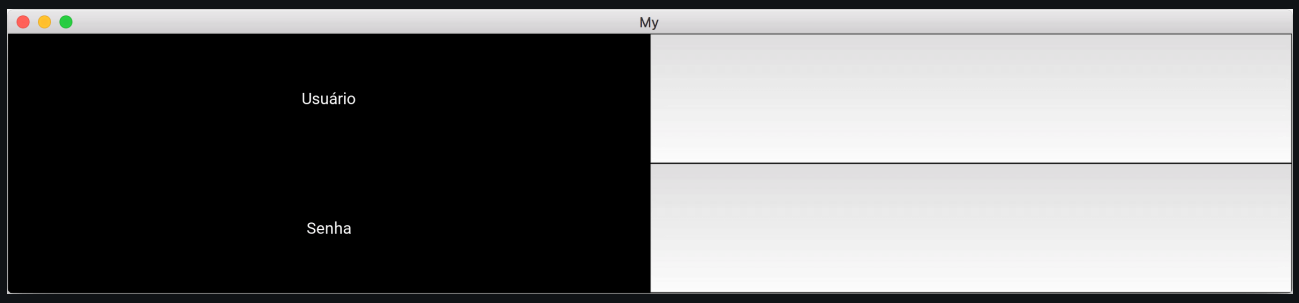
Um `Layout` é um elemento não renderizado (ele não aparece na tela) mas que serve para distribuir e posicionar elementos de maneiras distintas.

Os tipos de `Layout` do Kivy são: - `AnchorLayout`: Alinha os elementos dentro dele à uma das bordas (ou centro). - `BoxLayout`: Posiciona os elementos em "caixas" horizontais ou verticais. - `FloatLayout`: Todos os elementos dentro desse *layout* são posicionados absolutamente ao tamanho do *layout*. Para isso, esse *layout* honra as propriedades `pos_hint` e `size_hint`. - `GridLayout`: Posiciona os elementos em uma estrutura baseada em linhas e colunas (matriz). - `PageLayout`: É utilizado para criar *layouts* multi-página de uma forma que possibilite a navegação entre páginas utilizando as bordas do aplicativo. - `RelativeLayout`: Possibilita o posicionamento relativo dos elementos dentro desse *layout*. - `ScatterLayout`: Funciona similarmente ao `RelativeLayout` mas nesse *layout* também é possível movimentar, rotacionar e escalar (*scale*) usando toques ou cliques na tela. - `StackLayout`: Insere cada elemento verticalmente ou horizontalmente até o máximo que *layout* pode suportar. O tamanho de cada elemento filho não precisa ser uniforme.

Vamos escolher o `GridLayout` e vamos adicionar elementos de *input*.

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

class TelaLogin(GridLayout):
    def __init__(self, **kwargs):
        super(TelaLogin, self).__init__(**kwargs)
        self.cols = 2
        self.add_widget(Label(text='Usuário'))
        self.username = TextInput(multiline=False)
        self.add_widget(self.username)
        self.add_widget(Label(text='Senha'))
        self.password = TextInput(password=True, multiline=False)
        self.add_widget(self.password)

class MyApp(App):
    def build(self):
        return TelaLogin()

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

A saída é a seguinte:



Nesse exemplo, além do `GridLayout`, adicionamos também dois elementos de *input* chamados `TextInput` que serve para adquirir dados de texto do usuário.

A propriedade `password=True` diz ao Kivy que se trata de um campo de senha (portanto irá inserir **'*'*** nos caracteres inseridos) e a propriedade `multiline=False` significa que o texto será todo renderizado em uma linha.

> 💡 *Estou desenvolvendo o **DevBook**, uma plataforma que usa IA para gerar ebooks técnicos profissionais — com código formatado e exportação em PDF. Dá uma olhada!*

Arquivo de Configuração

Para facilitar nossa vida e customizar nossa aplicação conforme nossa necessidade, o Kivy possibilita que utilizemos um arquivo contendo configurações específicas por projeto.

A documentação do Kivy traz a lista completa de parâmetros que podemos incluir no nosso arquivo.

Por exemplo, se quisermos que a tecla `ESC` finalize a aplicação, podemos utilizar a configuração `exit_on_escape` com o valor de 1. Para isso, precisamos primeiro criar um arquivo de configuração, vamos chamar de `config.ini`, e adicionar o seguinte conteúdo:

```
[kivy]
exit_on_escape = 0
```

Algumas propriedades interessante que podemos utilizar:

- `window_icon = <string>`: caminho do ícone da janela em que a aplicação vai ser executado.
- `fullscreen = 0 ou 1`: Habilita/desabilita a aplicação em modo de tela cheia.
- `resizable = 0 ou 1`: Habilita/desabilita o redimensionamento da janela.
- `height = valor`: Altura da janela.
- `width = valor`: Comprimento da janela.

Com essas opções no nosso arquivo, podemos então fazer a leitura em nosso código com o objeto `kivy.config.Config` chamando o método `Config.read("config.ini")`.

Propriedades

Propriedades são uma forma incrivelmente fácil de definir eventos e vinculá-los ao nosso código.

Basicamente, propriedades produzem eventos que, havendo a alteração de um atributo de um objeto, todas as propriedades que referenciam aquele atributo são automaticamente atualizadas.

Para entedermos melhor, vamos comparar duas formas de se declarar propriedades. Antes teríamos:

```
class MinhaClasse(object):
    def __init__(self):
        super(MinhaClasse, self).__init__()
        self.valor_numerico = 1
```

Utilizando propriedades em Kivy, temos:

```
class MinhaClasse(Widget):
    valor_numerico = NumericProperty(1)
```

Não temos muito ganho em sintaxe. A maior diferença está no conceito por trás do objeto `Property`.

Propriedades implementam o Padrão de Projeto *Observer*.

Caso você não saiba, esse padrão de projeto cria dependências *um para muitos* entre objetos de modo que quando há uma alteração no estado de um objeto, todos os outros objetos dependentes são notificados automaticamente.

Dessa forma, propriedades nos ajudam a:

- Manipular facilmente *widgets* definidos na Linguagem Kivy.
- Observar qualquer mudança nos nossos objetos e responder automaticamente com algum método que irá tratar essa notificação.
- Checar e validar entradas.

- Otimizar o gerenciamento de memória da nossa aplicação já que o código de notificação do Kivy já foi testado e otimizado.

Para utilizarmos propriedades, devemos declará-las **no nível de classe**. Ou seja, não podemos declarar propriedades dentro de funções.

Abaixo estão listadas as propriedades definidas pelo Kivy:

- `StringProperty`: Representa uma String.
- `NumericProperty`: Representa um valor numérico.
- `BoundedNumericProperty`: Representa um valor numérico limitado por um valor mínimo e um valor máximo.
- `ObjectProperty`: Representa um objeto Python.
- `DictProperty`: Representa um `dict`.
- `ListProperty`: Representa uma lista.
- `OptionProperty`: Representa uma String dentro de uma lista de possíveis valores.
- `AliasProperty`: Caso você não ache nenhuma propriedade que satisfaça seu código, é possível criar uma propriedade customizada definindo um `getter` e um `setter`.
- `BooleanProperty`: Representa um booleano (`True` ou `False`).
- `ReferenceListProperty`: Permite a criação de tuplas de outras propriedades. Por exemplo, se tivermos uma `NumericProperty` chamada `id` e uma `StringProperty` chamada `nome`, podemos criar o seguinte objeto: `usuario = ReferenceListProperty(id, nome)`

Dessa forma, temos um conjunto completo de propriedades que podemos utilizar em nosso código!

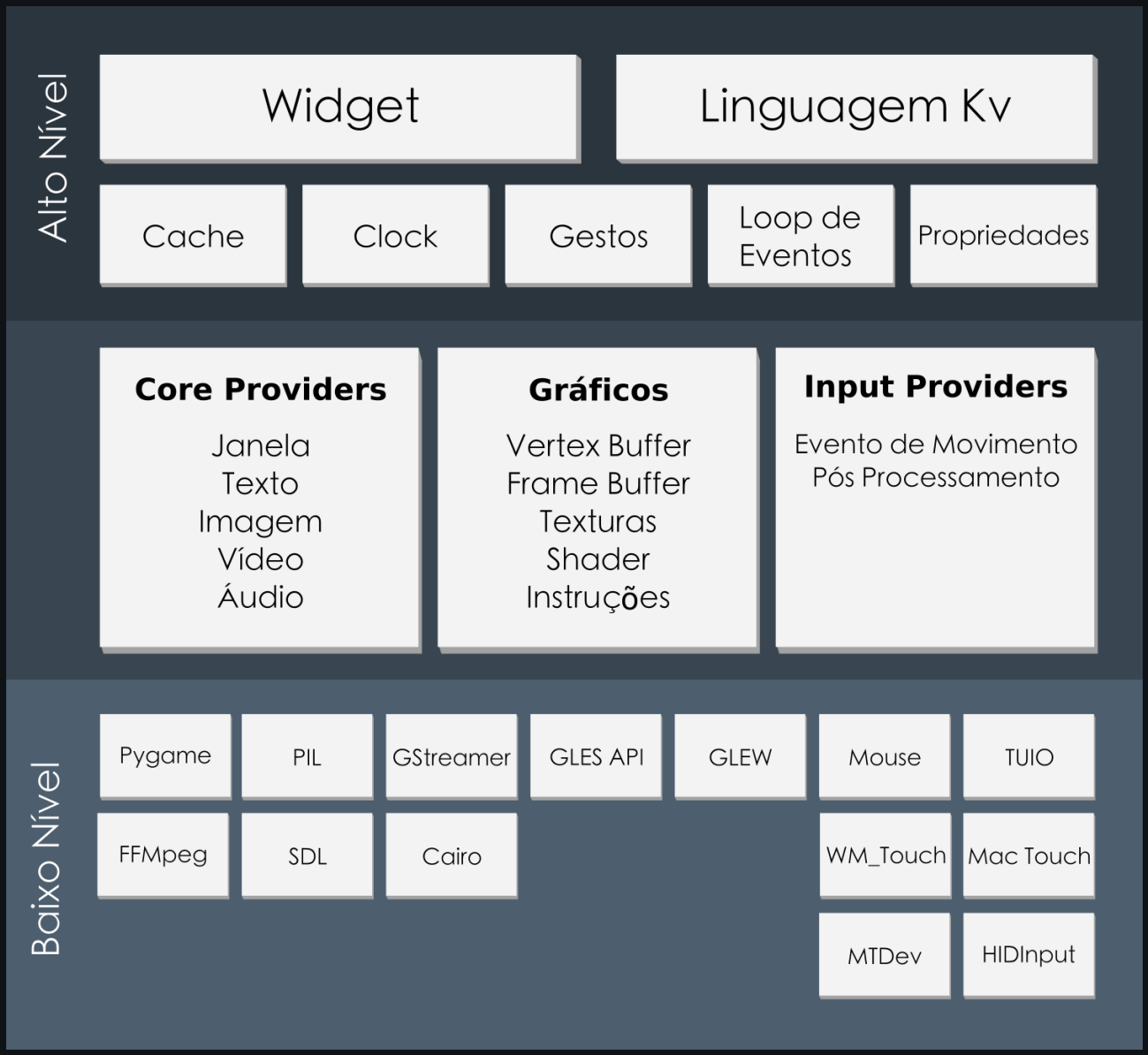
Vamos mergulhar agora na arquitetura do Kivy e entender melhor como as coisas funcionam por dentro.

Visão Geral da Arquitetura do Kivy

Entender como a arquitetura do Kivy foi desenhada ajuda e muito a compreender como os módulos internos funcionam em conjunto.

Essa seção explica a ideia básica por trás da implementação de cada módulo.

Para visualizarmos melhor a divisão e a interação entre os módulos, veja a ilustração abaixo:



Core Providers e Input Providers

Um conceito chave para entender a arquitetura do Kivy é sua modularidade e abstração.

O Kivy tenta abstrair simples tarefas, como abrir janelas, mostrar imagens e textos, tocar áudios e etc. Essas tarefas são chamadas de *core tasks* (tarefas de núcleo).

Isso torna a API do Kivy fácil de manter e fácil de utilizar.

E o mais importante, essa abstração possibilita ao Kivy utilizar o que chamam de *specific providers* (provedores específicos) para cada cenário em que sua aplicação poderá rodar (Mac OS, Windows, Linux, Android, iOS).

Por exemplo, no Mac OS, Linux e Windows, existem diferentes *APIs* nativas para cada *core task*.

Dessa forma, código que conversa com essas *APIs* nativas de um lado e com o Kivy do outro (atuando portanto como uma camada de comunicação), é chamado de *core provider*.

Isso torna o *framework* muito mais simples para nós programadores, pois não precisamos nos preocupar com aspectos específicos de cada sistema. Nós deixamos o Kivy fazer o que for melhor.

Também por conta disso, o pacote compilado do Kivy é bem menor (já que utiliza boa parte das funções de núcleo do sistema onde será executado).

O tratamento de *inputs* utiliza esse mesmo conceito. Um *input provider* é um conjunto de código que adiciona suporte à um dispositivo de entrada, como o *trackpad* da Apple ou um emulador de mouse, por exemplo.

Para adicionar um novo dispositivo, basta prover uma classe que realiza a leitura dos dados desse dispositivo e os transforma em eventos do Kivy.

Gráficos

A API gráfica provida pelo Kivy é uma abstração do OpenGL (*Open Graphics Library* - biblioteca padrão largamente utilizada no desenvolvimento de aplicações gráficas, ambientes 2D/3D, jogos).

No baixo nível, Kivy executa comandos do OpenGL acelerados por hardware utilizando o OpenGL.

Mesmo abstraindo esses detalhes para nós, o Kivy ainda possibilita que você utilize diretamente comandos do OpenGL.

UIX – *User Interface Experience* (*Layouts e Widgets*)

Neste módulo estão os elementos que utilizamos para compor nossa interface de usuário. e é composto, basciamente, por:

- `Widgets` são elementos que adicionamos ao nosso programa a fim de implementar alguma funcionalidade. Exemplos: botões, *sliders*, listas.
- `Layouts` são utilizados para combinar e organizar `Widgets` (vide seção acima).

Módulos

Módulos provêem uma maneira de customizar e adicionar funcionalidades à sua aplicação.

Como desenvolvedor, nós podemos tanto adicionar módulos do Kivy, quanto escrever nosso próprio módulo.

Alguns módulos presentes no Kivy são:

- `touchring`: Desenha um círculo ao redor de cada toque.
- `monitor`: Adiciona uma barra superior indicando o FPS (*Frame per Second*) e um pequeno gráfico indicando atividade de *input*.
- `keybinding`: Vincula alguns atalhos à ações, como por exemplo tirar um *screenshot* através de determinada tecla.
- `screen`: Emula as características de diferentes resoluções de tela.

Eventos de Entrada (Toques)

O Kivy também abstrai os diferentes tipos de *input* como toques, mouse e similares.

O que esses dispositivos têm em comum é que eles associam uma posição 2D em tela para um evento do Kivy.

Internamento no Kivy eles são representados como uma instância da classe `Touch`. Essa instância pode estar em um dos seguintes estados:

- ***Down***: Esse estado é ativado assim que a instância `touch` for ativada pela primeira vez (usuário clicou na tela, por exemplo).
- ***Move***: Esse estado ocorre quando a posição 2D é alterada (usuário movimenta o dedo na tela, por exemplo).
- ***Up***: É ativado apenas uma vez (quando o usuário retira o dedo da tela, por exemplo).

Widgets e Tratamento de Eventos

Um *widget* dentro do Kivy é um elemento que pode receber eventos de *input*.

Todo conjunto de *widgets* é representado internamente como uma árvore. Dessa forma, cada *widget* pode ter qualquer número de nós filhos e apenas um nó pai e todo *widget* é filho direta ou indiretamente do nó raiz (*root widget*).

Quando um evento de toque é acionado, ele é enviado ao *widget* raiz o qual pode processar o evento ou passá-lo adiante. Caso algum tratador de evento (*event handler*) retorne `True`, o evento é dado como processado e tratado corretamente.

Esse processo ocorre até que o evento chegue à base da árvore de *widgets*.

Linguagem Kivy (*kvlang* ou *Kivy Language*)

Agora que já temos uma noção básica de como o Kivy funciona, vamos abordar um conceito extremamente importante que ajuda muito na organização da nossa aplicação: a linguagem Kivy.

Conforme desenvolvemos, nossa aplicação vai se tornando cada vez mais complexa. Nossa estrutura de *widgets* vai crescendo e se tornando mais difícil de manter.

Nesse sentido, o pessoal do Kivy desenvolveu uma linguagem interna para descrição da nossa árvore de *widgets*. Essa linguagem possibilita vincular propriedades e/ou funções de *callback* entre *widgets*.

Essa abordagem possibilita a criação de protótipos e atualizações na interface de usuário de maneira eficiente.

Além disso facilita a separação entre a lógica da nossa aplicação e a interface de usuário.

Carregando seu Arquivo .kv

Podemos carregar nosso arquivo .kv de duas formas: - **Por convenção**: O Kivy procura por um arquivo com o mesmo nome da sua classe principal (aquela que estende de `App`) em letras minúsculas e sem o sufixo "App", caso tenha. Exemplo: sua classe se chama `MyFirstApp`. Assim, o arquivo buscado será **myfirst.kv**. - **Utilizando a classe `Builder`**: Nós podemos forçar o Kivy a carregar os dados de um arquivo de nossa escolha com o método `load_file('arquivo.kv')` da classe `Builder`.

Contexto de Regras (*rules*)

A linguagem Kivy é baseado em regras, que descrevem o conteúdo de um *widget*.

Em um arquivo .kv existe apenas uma regra raiz (*root rule*) e qualquer número de regras filhas. Essa regra raiz é definida pelo nome da regra, sem identação, seguida de ':'. Exemplo:

```
Widget:
```

Uma regra de classe (*class rule*) é definida pelo nome da regra delimitada por "<>" seguida de ":". Exemplo:

```
<WidgetClass>:
```

Algumas construções que podem ser feitas: - Acessar módulos Python e classes do Kivy:

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

que é equivalente à:

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

em python.

- Definir uma variável global:

```
#:set name value
```

e em Python:

```
name = value
```

Declaração de *Widgets* Filhos

Para declarar que determinado *widget* tem um filho, precisamos apenas defini-lo dentro da respectiva regra. Exemplo, se quisermos definir um *widget* com um `BoxLayout` que tem dois botões, podemos fazer assim:

```
WidgetRaiz:
    BoxLayout:
        Button:
        Button:
```

Para passar parâmetros para um *widget* (assim como fazemos em Python com `grid = GridLayout(cols=3)`), fazemos:

```
GridLayout:
    cols: 3
```

Bind de Eventos

Podemos realizar o *bind* de funções à eventos da seguinte forma:

```
Widget:
    on_size: callback()
```

Dessa forma, quando o evento `on_size` por processado, Kivy chamará a função `callback()`.

Acessando *Widgets* Definidos no Arquivo .kv em seu Código Python

Bem, até agora, citamos diversos maneiras de construir uma aplicação utilizando Kivy.

Agora precisamos definir como acessar os componentes definidos no arquivo .kv no código Python.

Para podermos referenciar componentes Kivy em nosso código, utilizamos identificadores (`id`) para tal.

Por exemplo:

```
<Widget>:
    Button:
        id: button_id
    TextInput:
        text: button_id.state
```

Dessa forma, `TextInput` está referenciando o elemento `Button` pelo seu identificador `button_id`.

Uma observação importante: `id` é uma referência fraca e portanto não evita que o elemento seja coletado pelo `garbage collector`.

Quando seu seu arquivo .kv é processado, o Kivy coleta todos os elementos e monta um dicionário `self.ids` contendo os identificadores lidos.

Veja o código Kivy abaixo:

```
<WidgetExemplo>
    Label:
        id: email
        text: 'Email'
    Button:
        id: submit
        text: "Enviar"
        on_release: root.submit_email()
```

Um código Python que acessa esses elementos:

```
class WidgetExemplo(BoxLayout):
    def submit_email(self):
        self.ids.submit.text = "Email enviado"
        self.ids["email"].text = ""  # Sintaxe alternativa
```

Dessa forma, temos o *link* entre o código Kivy e o Python!

Classes Dinâmicas

Classes dinâmicas servem para criar *templates* padrão para reutilizar partes repetidas entre componentes.

Por exemplo, considere o código abaixo:

```
<MeuWidget>:
    Button:
        text: "Este texto está dentro de um botão"
        text_size: self.size
        font_size: '32sp'
        markup: True
    Button:
        text: "Outro texto dentro de outro botão"
        text_size: self.size
        font_size: '32sp'
        markup: True
    Button:
        text: "Mais uma vez temos um texto dentro do botão"
        text_size: self.size
        font_size: '32sp'
        markup: True
```

Percebe que as propriedades `text_size`, `font_size` e `markup` se repetem dentro de cada componente?

Para evitar isso, podemos criar um *template* para criar partes reutilizáveis de código. Exemplo:

```
<BotaoGrande@Button>:
    text_size: self.size
    font_size: '32sp'
    markup: True
```

Para estender esse *template*, desenvolvemos da seguinte forma:

```
<MeuWidget>:
    BotaoGrande:
        text: "Este texto está dentro de um botão"
    BotaoGrande:
        text: "Outro texto dentro de outro botão"
    BotaoGrande:
        text: "Mais uma vez temos um texto dentro do botão"
```

Assim evitamos código duplicado e aumentamos a flexibilidade de nosso código, podendo criar componentes que extendem outros.

Outra forma de reutilização de código é reutilizar a mesma definição da classe para diferentes propósitos. Por exemplo:

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

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

Os dois elementos possuem o mesmo estilo. Dessa forma, podemos reutilizar seus códigos da seguinte forma:

```
<PrimeiroWidget, SegundoWidget>:
    Button:
        on_press: root.text(txt_inpt.text)
    TextInput:
        id: txt_inpt
```

Assim, ambas classes compartilharão o mesmo código Kivy.

Conclusão

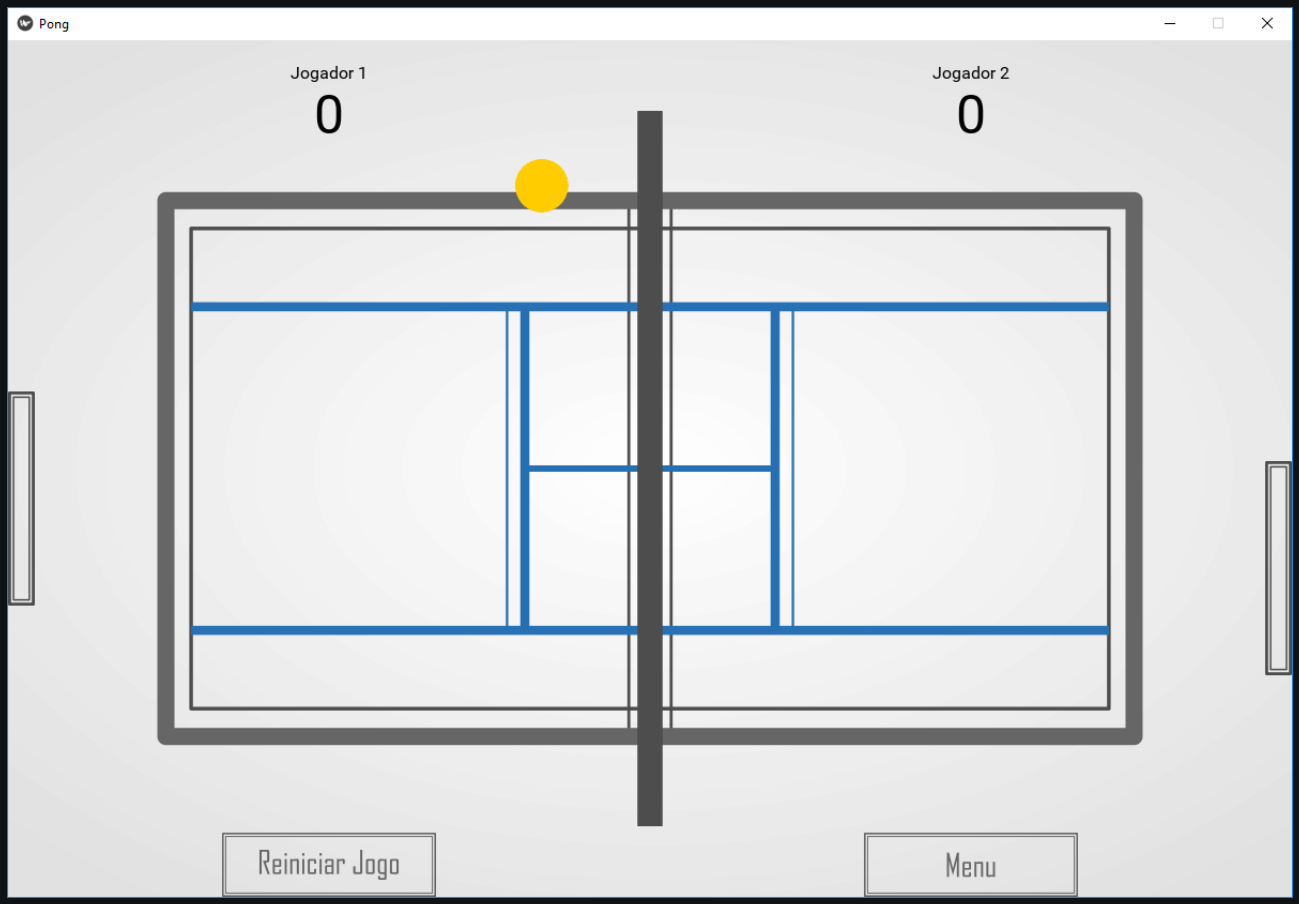
Nesse artigo vimos um pouco do Kivy e como podemos começar a utilizá-lo.

Vimos alguns conceitos importantes, como sua Arquitetura e a Linguagem Kivy, os quais nos ajudam a compreender o funcionamento das nossas aplicações feitas utilizando esse poderoso *framework*.

Vimos que é possível desenvolvermos aplicações simples com poucas linhas de código.

Espero ter despertado a curiosidade de vocês com relação ao Kivy! Esse foi apenas um post introdutório.

No segundo artigo, explico alguns conceitos importantes, como o gerenciamento de telas, áudio, funções periódicas com *Clock*, botões customizados e **muito mais**!!! Também construo o seguinte jogo, chamado Pong Reborn:



Ele é uma recriação do famoso jogo Pong. Mas na nossa versão de qualidade! 😉

Pra não perder o embalo, já acesse o segundo artigo clicando aqui!

Ah, e não se esqueça de comentar qualquer dúvida ou sugestão aqui embaixo! Seu feedback é **MUITO** importante para nós!

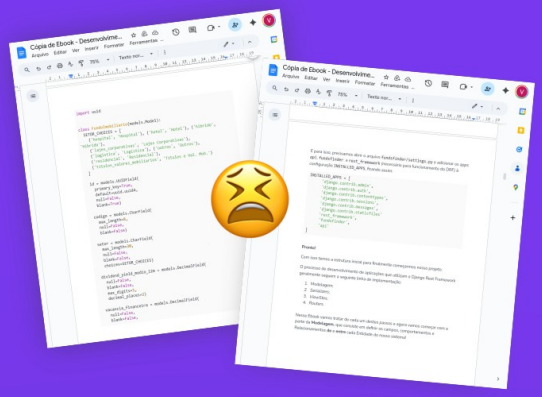
Bom desenvolvimento!

Não se esqueça de conferir!

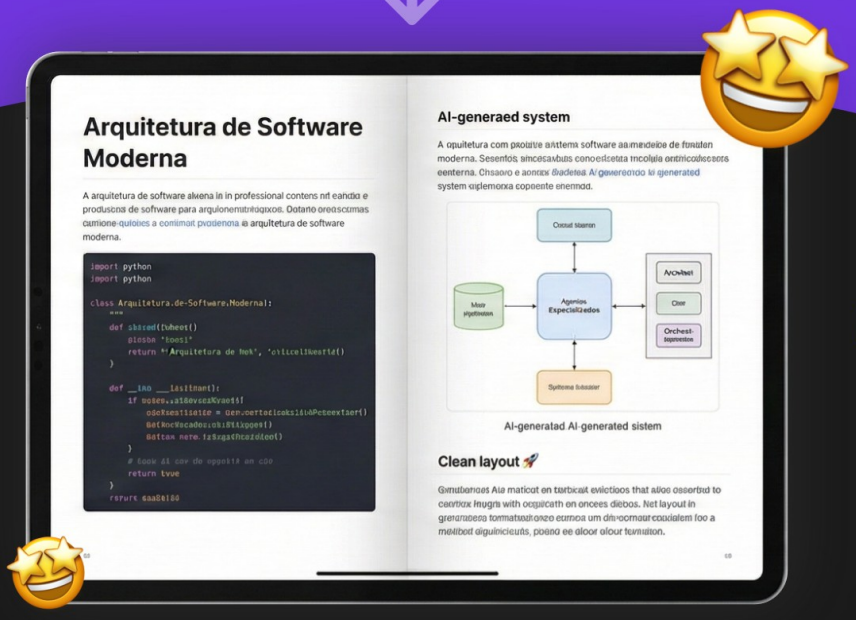
DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS