PYTHON ACADEMY

CRIE JOGOS COM KIVY E PYTHON

Que tal criar um jogo em Python utilizando o framework Kivy? Nesse ebook vamos falar sobre menus, telas, áudio e desenvolver um jogo completamente do ZERO!

PYTHONACADEMY.COM.BR

No post anterior, nós discutimos as principais características do Kivy, uma ferramenta muito poderosa para o desenvolvimento de aplicações em Python.

Agora que você já tem uma boa noção sobre as capacidades do Kivy, chegou a hora de colocar a mão na massa, ou melhor, no código, e usá-lo para fazer algo mais complexo. Que tal um jogo? Acha uma boa ideia? 😉

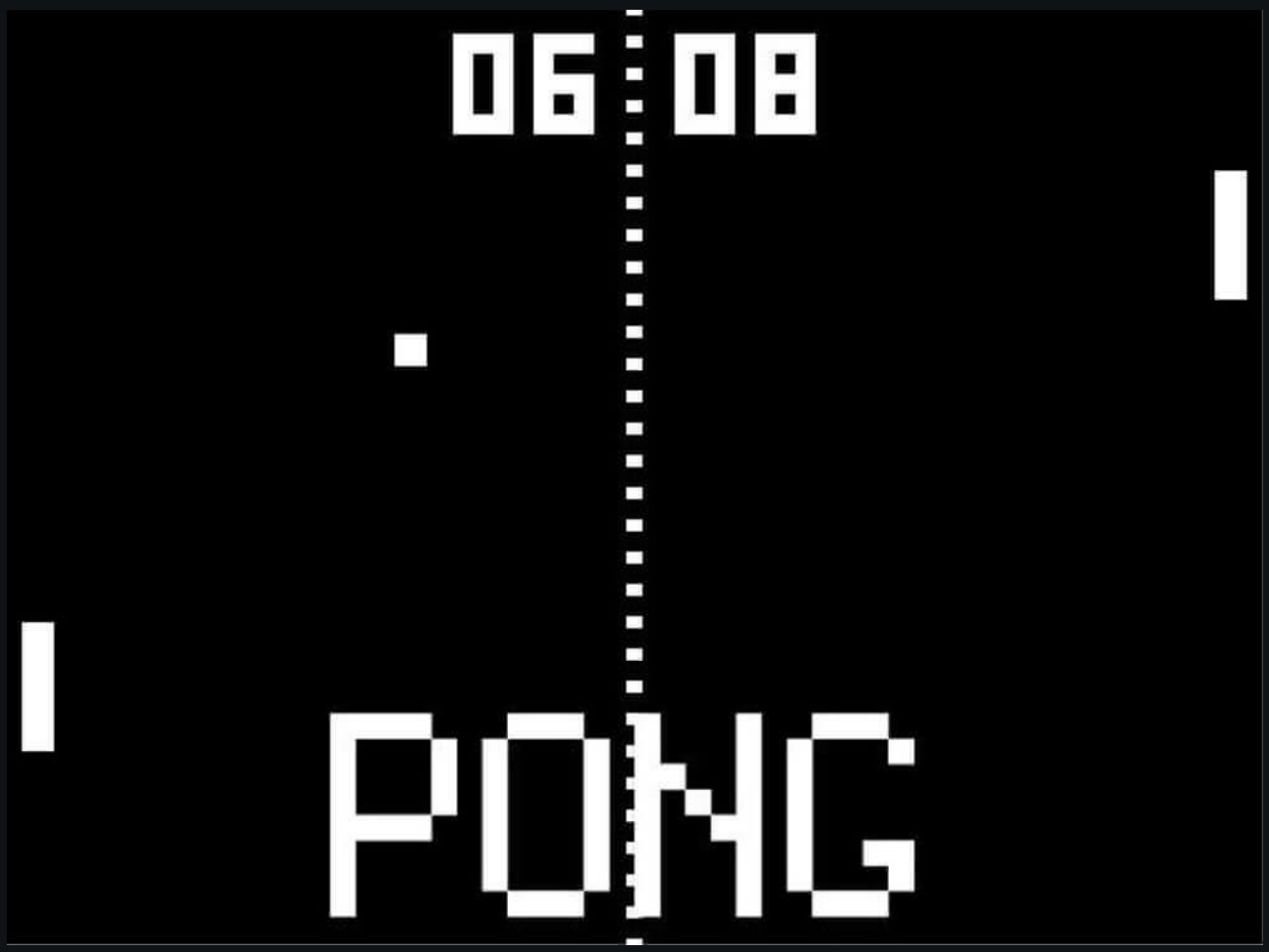
Sem mais delongas, neste *post* nós iremos:

- Desenvolver uma aplicação gráfica baseada no jogo Pong.
- Dividir a nossa aplicação em classes.
- Aprender recursos fundamentais do Kivy (ex: gerenciamento de telas, inclusão de menus, inserção de áudio, e muito mais).

{% raw %}#Partiu!{% endraw %}

Pong Reborn!

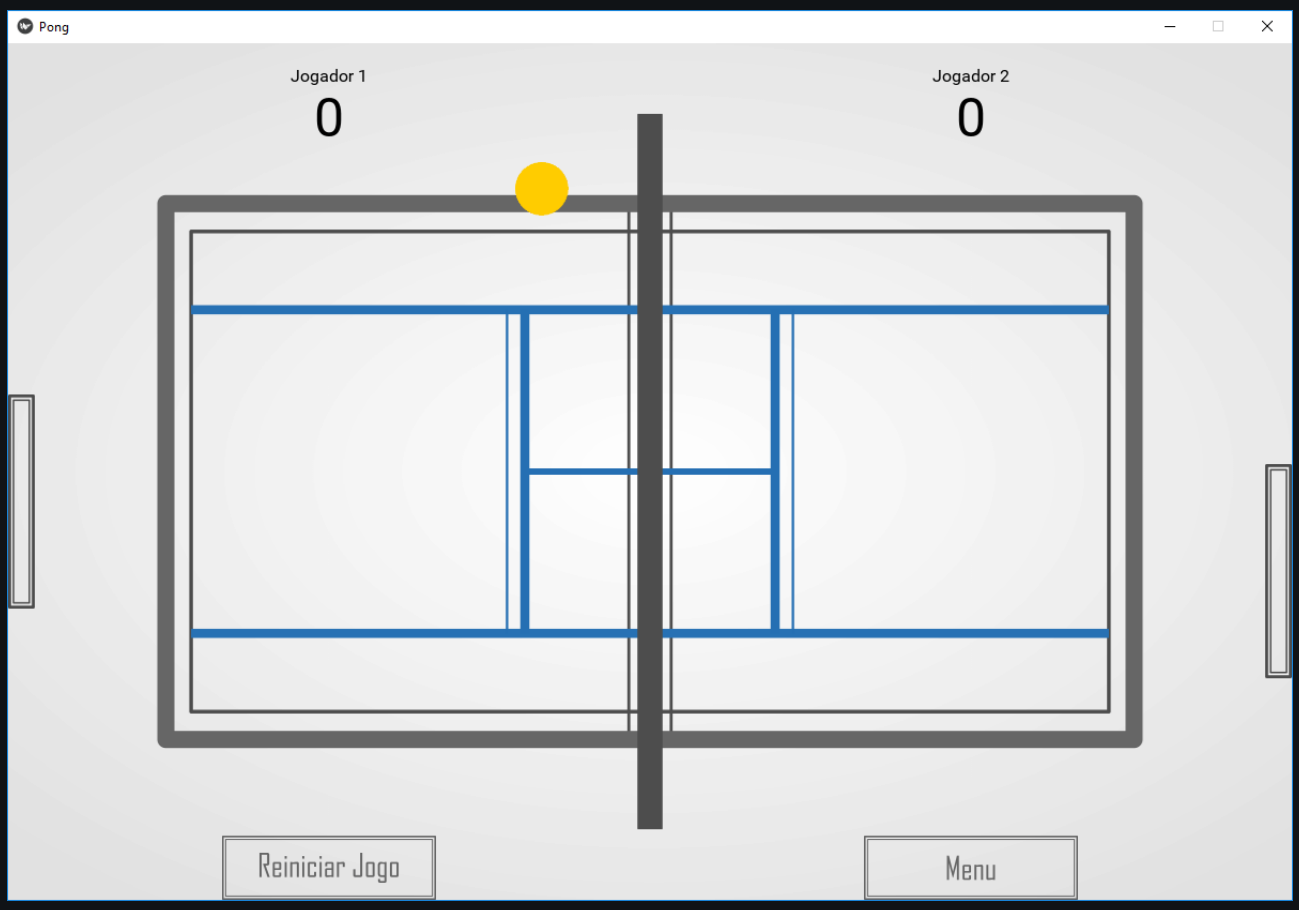
Nós já aprendemos no *post* anterior a estruturar nossas aplicações com o Kivy, então vamos direto para o desenvolvimento de uma aplicação real: o famoso jogo Pong!



É claro que nossa versão do Pong! não poderia ser tão simples como o jogo acima… Demos uma repaginada e criamos uma “versão contemporânea” do jogo: o Pong Reborn!

Nele desenvolveremos as telas, os menus, adicionaremos áudio e **MUITO MAIS**!!!

Nosso objetivo final é criar a seguinte aplicação:



Aquecendo os Motores

Primeiro, vamos criar a estrutura de diretórios do projeto:

Nela, temos:

- `/audio` : contém os arquivos de áudio do jogo.
- `/img` : diretório para guardar as imagens (raquete, bola, botões, etc).

- `/telas` : contém as classes de definição das telas do nosso jogo.
- `/widgets` : contém os *widgets* necessários.
- `main.py` : código de inicialização do nosso projeto
- `pong.kv` : arquivo .kv contendo o *layout* da nossa aplicação na linguagem Kivy.
- `config.ini` : arquivo de configuração com parâmetros do nosso jogo.

Para progredirmos juntos, baixe aqui o código completo da aplicação que será produzida nesse *post*. Meu conselho é que você replique a estrutura e desenvolva conforme vai lendo aqui no *post* e, tendo qualquer dúvida, dá uma espiada no código. Lembrando que você sempre tem a opção de deixar as suas dúvidas na seção de comentários ali embaixo…

Não se preocupe que cada linha do código será explicada aqui.

Vamos começar iniciando uma aplicação simples para servir de base para o nosso jogo.

```
from kivy.uix.widget import Widget
from kivy.app import App

class Pong(Widget):
    pass

class PongApp(App):
    def build(self):
        return Pong()

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

Se seu ambiente de desenvolvimento estiver configurado corretamente, uma tela preta deverá aparecer ao executar essa aplicação.

Carregando o Arquivo de Configurações

Assim como vimos no *post* anterior, podemos utilizar um arquivo de configurações para passar parâmetros de inicialização para a nossa aplicação.

Coloque os seguintes parâmetros no arquivo `config.ini`:

```
[kivy]
exit_on_escape = 1
[graphics]
resizable = 0
width = 1200
height = 800
```

Eles nos dizem que:

- `exit_on_escape = 1`: a tecla `ESC` encerrará a aplicação.
- `resizable = 0`: não deve ser possível redimensionar a janela da aplicação.
- `width = 1200`: largura da janela.
- `height = 800`: altura da janela.

Para carregar o arquivo de configurações, utilizamos o método `read("config.ini")` da classe `kivy.config.Config`:

```
from kivy.config import Config

Config.read("config.ini")

class Pong(Widget):
    pass

...
```

Como o arquivo `config.ini` já está na pasta raiz do projeto, não precisamos utilizar o caminho absoluto para ele.

Gerenciando as Telas com o `ScreenManager`

Uma preocupação que devemos ter quando desenvolvemos aplicações com as quais os usuários irão interagir é **navegação** e a **usabilidade**.

Usuários estão acostumados a ver *links* agrupados em menus, transições entre telas, botões de navegação, etc.

Felizmente, o Kivy provê formas de controlar a navegação do usuário através da definição de telas (`Screen`) e de métodos da classe `ScreenManager` .

Por exemplo, para criarmos um gerenciador `ScreenManager` com duas telas, podemos fazer:

```
from kivy.uix.screenmanager import ScreenManager, Screen

# Create the manager
manager = ScreenManager()

# Criação das telas
tela_1 = Screen(name='Tela 1')
tela_2 = Screen(name='Tela 2')

# Adiciona as telas ao ScreenManager
manager.add_widget(tela_1)
manager.add_widget(tela_1)

# Por padrão, a primeira tela adicionada será aquela mostrada pelo ge-
        renciador.
# Para mudar para, por exemplo, a tela 2, utilizamos o parâmetro name:
manager.current = 'Tela 2'

# Nesse momento, a Tela 2 será mostrada!
```

Nesse caso, as telas estarão em branco (preto, na verdade), pois não adicionamos nada a elas.

Podemos criar uma classe vazia e definir o *layout* da nossa tela em nosso arquivo `.kv`.

Como exemplo, vou pegar uma tela de vencedor definida na nossa aplicação:

```
# Declara a Tela do Vencedor 1
class TelaVencedor1(Screen):
    pass
```

E seu *layout* será:

```
<TelaVencedor1>:
    canvas:
        Rectangle:
            source: 'img/vencedor-1.png'
            size: self.width, self.height

    Button:
        size_hint: 0.2, 0.1
        pos_hint: {'top': 0.5, 'right': (0.7 + (self.width / 2) /
        root.width)}
        background_normal: 'img/voltar-ao-menu-btn.png'
        background_down: 'img/voltar-ao-menu-btn.png'
        center_x: root.center_x + self.width / 2 + 20
        on_press:
            root.manager.current = "menu"
            root.manager.transition.direction = "right"
```

Esse *layout* traz alguns conceitos novos:

- Podemos definir uma imagem como *background* utilizando a propriedade `canvas` e o elemento `Rectangle` com seu parâmetro `source` sendo uma imagem.

- Para utilizarmos uma imagem de fundo no *widget* `Button`, podemos usar as propriedades `background_normal` e `background_down`, ambas apontando para o caminho onde estão as imagens (`/img` no nosso caso).
- Para definir o comportamento esperado quando clicarmos no botão, podemos utilizar o evento `on_press` ("quando clicado"). Nesse caso, o botão está realizando uma transição de telas, pedindo ao `manager` para mudar para a tela `menu`, com uma transição da esquerda para direita.

Você pode conferir as outras telas da aplicação no arquivo `pong.kv`.

Lembrando: **qualquer dúvida**, postem aqui embaixo nos comentários!

Agora que sabemos gerenciar as telas, podemos descrever os elementos que farão parte do jogo.

> 💡 *Estou construindo o **DevBook**, uma plataforma que usa IA para criar ebooks técnicos — com código formatado e exportação em PDF. Depois de ler este artigo, dá uma passada lá!*

Adicionando Áudio com `SoundLoader`

O que seria de um jogo sem áudio? Já imaginou o que seria de **Top Gear** ou **F-Zero** sem suas incríveis trilhas sonoras?

Vamos então colocar áudio em nossa aplicação.

Primeiro devemos escolher uma trilha. Para isso, temos três opções:

1. Criamos nossa própria trilha (infelizmente não tenho essa habilidade… **Ainda**!)
2. Compramos uma trilha
3. Arranjamos uma trilha 0800 na “net”

Utilizei a última opção! Aqui vão alguns sites em que busquei:

- digCCmixter
- SampeSwap

Para nossa aplicação, escolhi uma trilha similar àquelas encontradas em *indie games*... Achei bem legal!

Com sua trilha em mãos, basta colocá-la na pasta `/audio` e carregá-la com o método `load()` da classe `kivy.core.audio.SoundLoader`.

Feito isso, podemos colocá-la em execução com o método `play()`.

O código fica assim:

```
# Carrega o áudio
sound = SoundLoader.load('audio/bg-music.mp3')

# Verifica se houve o carregamento do nosso áudio e o coloca em exeução
if sound:
    sound.loop = True
    sound.play()
```

Para que o áudio volte ao começo quando terminar, configuramos o parâmetro `loop` do objeto gerado pela classe `SoundLoader` com `True`.

Adicionando Elementos Gráficos

Em nosso jogo, temos três elementos básicos: a **raquete**, a **bola** e o **campo**.

Teremos duas raquetes, uma de cada lado do campo, que se moverão pelo arrastar do mouse (ou do dedo).

A bola estará em constante movimento, indo de um lado para o outro de acordo com o toque na raquete ou nos lados do campo (significando ponto para um jogador).

Por último, temos o campo, onde estarão os limites do nosso jogo. Nesse elemento, estarão a raquete e a bola, já que elas “precisam” de um campo para fazer sentido.

Classe `Raquete`

Primeiro, vamos descrever o *widget* **raquete**.

Esse elemento tem apenas duas obrigações dentro da lógica do nosso jogo: manter o placar e descrever o comportamento de como rebater a bola, ou seja, qual deve ser a nova direção da bolinha ao tocar na raquete.

Assim, nossa classe `Raquete` será implementada da seguinte forma:

```
from kivy.properties import NumericProperty
from kivy.uix.widget import Widget
from kivy.vector     import Vector

class Raquete(Widget):
    placar = NumericProperty(0)

    def rebate_bola(self, bola):
        if (self.collide_widget(bola)):
            vx, vy = bola.velocidade

            offset_raquete = (bola.center_y - self.center_y) /
        (self.height / 2)

            inv_vel = Vector(-1 * vx, vy)

            vel = inv_vel * 1.15

            bola.velocidade = vel.x, vel.y + (offset_raquete * 2)

        return
```

Nela, temos o `placar`, que é uma propriedade numérica que começa em 0 (`NumericProperty(0)`), e o método que controla o rebote da raquete.

O código do método `rebate_bola` pode parecer complicado, então vamos explicá-lo linha a linha:

- **Linha 10**: Verifica se houve a colisão do *widget* "raquete" com o *widget* "bola".
- **Linha 11**: Pega a tupla da velocidade da bola `(vx, vy)`. Esse atributo vai ser explicado quando falarmos da classe `Bola`.
- **Linha 13**: Verifica se a bola bateu na parte de cima ou embaixo da raquete. O atributo `center_y` da bola traz a altura "y" da bola com relação à altura total da janela e `height` é a altura do *widget*.

- **Linha 15**: Inverte a direção da bola. A classe `Vector` do Kivy representa um vetor 2D com suas componentes "x" e "y". É implementado como uma lista em Python, então podemos seus elementos com `x = vetor[0]` e `y = vetor[1]`.
- **Linha 17**: Aumenta a velocidade da bolinha. Deixo aqui como sugestão você brincar com esse valor (que está atualmente como "1.15").
- **Linha 19**: Atualiza a direção e velocidade da bolinha, fazendo-a subir ou descer (dependendo de onde ela tenha pegado na raquete: em cima ou embaixo).

Classe `Bola`

Essa classe mantém o estado da velocidade, sentido e direção da bola no jogo.

A componente da velocidade no eixo X vai ser representada pela variável `velocidade_x` e a componente da velocidade no eixo Y vai ser representada pela variável `velocidade_y`.

Para facilitar nossa vida posteriormente, definiremos também uma variável chamada `velocidade` que será uma tupla contendo as duas componentes acima citadas.

Por fim, precisaremos apenas codificar o movimento da bolinha no campo (já que a verificação de colisão com a raquete já foi implementada na classe `Raquete`).

Uma forma é atualizar a propriedade `pos` (de `position`) do nosso *widget* bola com o vetor velocidade.

Juntando tudo, chegamos à seguinte implementação:

```
from kivy.uix.widget import Widget
from kivy.vector     import Vector
from kivy.properties import NumericProperty, ReferenceListProperty

class Bola(Widget):
    velocidade_x = NumericProperty(0)
    velocidade_y = NumericProperty(0)
    velocidade = ReferenceListProperty(velocidade_x, velocidade_y)

    def movimenta(self):
        self.pos = Vector(*self.velocidade) + self.pos
        return
```

Agora só falta definir o campo, que chamaremos de `Pong`.

Classe `Pong`

A responsabilidade dessa classe é manter o controle sobre todo o jogo.

A classe `Pong` conterá as outras classes e também verificará colisões entre a bola e as paredes laterais, a fim de atualizar o placar do jogo.

Assim, precisamos de três variáveis:

- `bola`: referencia a bola dentro do jogo.
- `raquete_1` e `raquete_2` referenciam as raquetes.

Também precisamos de um método (ou serviço) para colocar a bola em jogo. Esse método consiste basicamente em colocar a bola no centro do jogo e configurar sua direção (direita ou esquerda) dependendo de quem está servindo o ponto.

Finalmente, chegamos ao seguinte código:

```
def servico(self, vel=(4, 0)):
    # Posiciona a bola no centro da tela
    self.bola.center = self.center

    # Seta a velocidade da bola
    self.bola.velocidade = vel
```

Também desenvolveremos um método responsável por atualizar todo o jogo, *frame* a *frame*, movimentando elementos, verificando colisões e atualizando o placar. A lista de responsabilidades desse método é a seguinte:

- Movimentar a bola pelo campo.
- Rebater a bola caso haja a colisão entre a raquete e a bola.
- Verificar se a bola atingiu o topo ou o fundo da janela, a fim de evitar que a bola suma da janela.
- Verificar se houve colisão com o lado esquerdo ou o lado direito da janela, para atualizar o placar do jogo.
- Quando um dos jogdores chegar ao placar máximo (5, no nosso caso), exibir a tela de “Parabéns” (vamos falar sobre o `ScreenManager` já, já).

O código para fazer tudo isso é o seguinte:

```
# Atualiza o jogo
def atualiza(self, dt):
    # Faz a bola se mover
    self.bola.movimenta()

    # Rebate a bola caso haja colisão com a bolinha
    self.raquete_1.rebate_bola(self.bola)
    self.raquete_2.rebate_bola(self.bola)

    # Verifica se a bola atingiu o topo da janela
    if (self.bola.y < 0) or (self.bola.top > self.height):
        self.bola.velocidade_y *= -1

    # Verifica se colidiu com o lado esquerdo da janela para atualizar
        o
    # placar do jogo
    if self.bola.x < self.x:
        # +1 para o placar da raquete_2
        self.raquete_2.placar += 1

        if self.raquete_2.placar >= 5:
            self.servico(vel=(0, 0))
            self.raquete_1.placar = 0
            self.raquete_2.placar = 0
            self.screen_manager.current = "vencedor_2"

            return

        # Reinicia o jogo com a bola saindo pelo lado esquerdo
        self.servico(vel=(4, 0))

    # Verifica se colidiu com o lado direito da janela para atualizar o
    # placar do jogo
    if self.bola.x > self.width:
        # +1 para o placar da raquete_1
        self.raquete_1.placar += 1

        if self.raquete_1.placar >= 5:
            self.servico(vel=(0, 0))
            self.raquete_1.placar = 0
            self.raquete_2.placar = 0
```

```
            self.screen_manager.current = "vencedor_1"

            return

        # Reinicia o jogo com a bola saindo pelo lado direito
        self.servico(vel=(-4, 0))
```

Outras duas funções importantes são a `comeca_jogo()` e `reinicia_jogo()`... 😉

Começar o jogo é, basicamente, colocar a bola em jogo (aqui podemos chamar o método `servico()`) e configurar qual método será periodicamente chamado no *loop* principal.

Isso pode ser feito com o método `schedule_interval` da classe `kivy.clock.Clock`, passando-se como argumento o método de atualização (`atualiza()`) e a periodicidade com que ele deve ser executado pelo Kivy.

O código da `comeca_jogo()` é o seguinte:

```
def comeca_jogo(self):
    # Põe a bola em jogo
    self.servico()

    # Agendamento do método "atualiza" a cada 1/120 = 0,008s
    Clock.schedule_interval(self.atualiza, 1.0/120.0)
```

Já a `reinicia_jogo()` apenas põe a bola em jogo, a partir do método `servico()`, e zera o placar:

```
def reinicia_jogo(self):
    # Põe a bola em jogo
    self.servico(vel=(4,0))

    # Zera o placar
    self.raquete_1.placar = 0
    self.raquete_2.placar = 0
```

Agora só falta modelar o método para movimentar as raquetes conforme arrastamos o mouse ou o dedo na tela.

Para isso, o Kivy disponibiliza algumas funções na classe `Widget`. São elas:

- `on_touch_down()`: Evento disparado quando o Kivy identifica um toque na tela.
- `on_touch_move()`: Evento disparado quando o usuário arrasta o dedo ou o *mouse* na tela (estamos interessados nesse evento).
- `on_touch_up()`: Disparado quando o usuário estava solta determinada tecla.

Com o evento `on_touch_move()` podemos verificar a posição "X" do toque para saber qual raquete foi arrastada. Caso `touch.x` seja menor que a largura dividida por 2 (`self.width / 2`), devemos arrastar a raquete da esquerda. Caso `touch.x` seja maior que a metade do campo, arrastamos a raquete da direita.

Segue o código:

```python
# Captura o evento on_touch_move (arrastar de dedo na tela)
def on_touch_move(self, touch):
    # Verifica se toque foi do lado esquerdo da tela
    if touch.x < self.width / 2:
        # Atualiza altura da raquete esquerda
        self.raquete_1.center_y = touch.y

    # Verifica se toque foi do lado direito da tela
    if touch.x > self.width - self.width / 2:
        # Atualiza altura da raquete direita
        self.raquete_2.center_y = touch.y
```

Bom, com isso, acho que percorremos boa parte do código. Agora cabe a você explorar o que ficou faltando! 😎

Lembrando que **qualquer** dúvida é sempre muito bem vinda na nossa seção de discussão aqui embaixo!

Ah, e mais uma vez, caso ainda não tenha entrado para a lista de e-mails mais Pythônica do Brasil! 👇

Espero que você tenha aprendido e curtido bastante! 👌

Conclusão

O Kivy é uma ferramenta bastante poderosa na criação de interfaces gráficas! Principalmente pelo fato de poder contar com a simplicidade do Python.

O que eu não achei interessante foi a forma de descrever o *layout* de maneira separada, nos arquivos `.kv`. Achei que alguns elementos ficaram soltos, como por exemplo configurar a direção da transição utilizando o `root.manager.transition.direction`.

Outra desvantagem é a interface padrão de botões, campos de entrada, *tabs* entre outros que o Kivy nos proporciona.

Com relação à documentação, achei ela boa, mas apenas no “caminho feliz”. Quando as coisas apertam, é preciso recorrer ao nosso bom e velho StackOverflow.

Achei difícil comparar o Kivy com outros *frameworks* mais maduros que encontramos no mercado, como por exemplo, o Ionic, que traz a facilidade de utilizar HTML/CSS/Javascript para desenvolvimento de aplicativos móveis e PyQT para aplicações *desktop*.

Em resumo, gostei de desenvolver utilizando Kivy. O código é bem estruturado e fácil de desenvolver, muito por conta do Python. Mas fazer algo bonito no Kivy é uma tarefa bastante complicada! 😔

É isso aí pessoal!

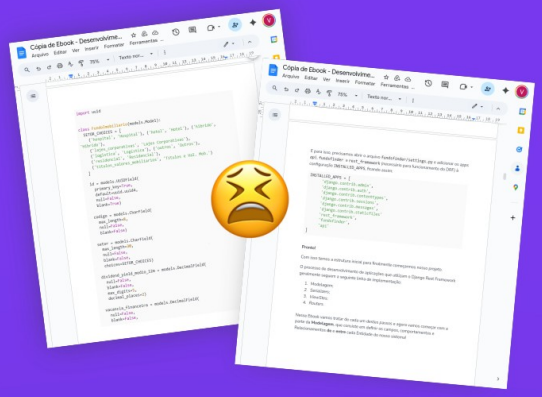
Nos vemos no próximo *post*! 😁

Não se esqueça de conferir!

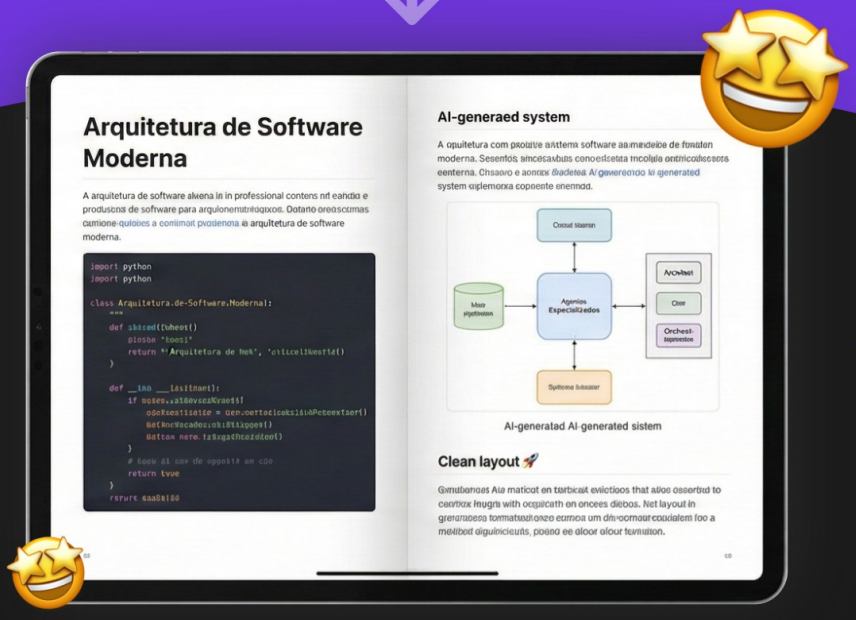
DevBook

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