

# Desenvolvimento de Aplicativos Mobile



## Boas Vindas!

Em um mundo cada vez mais conectado, a palma da sua mão tornou-se uma janela para o universo digital. Através dos *smartphones*, interagimos com amigos, fazemos compras, jogamos, aprendemos e até trabalhamos. Os aplicativos móveis são o coração dessa revolução, tornando tudo isso possível. E é justamente essa magia da criação que você está prestes a descobrir e dominar neste curso.

A jornada que você está prestes a trilhar abrangerá desde os primórdios dos PDAs e *smartphones* primitivos, passando pela evolução e sofisticação dos Sistemas Operacionais móveis, até os desafios e oportunidades no desenvolvimento de aplicativos nativos e híbridos. Juntos, vamos explorar as nuances do *design de interfaces* que encantam, a arquitetura de aplicações resilientes e a integração com o vasto ecossistema de serviços Web e APIs.

A relevância dos temas que abordaremos não é apenas histórica ou teórica. Cada tópico foi cuidadosamente escolhido para refletir as habilidades e conhecimentos que o mercado de trabalho exige atualmente. Ao compreender a essência da programação para *smartphones*, você estará equipado não apenas para desenvolver aplicativos, mas para criar experiências que poderão impactar milhões de usuários ao redor do mundo.

Sabemos que todo aprendizado tem seus desafios. Haverá momentos de incerteza e talvez até de frustração. Mas lembre-se: cada desafio é uma oportunidade de crescimento. Então, convidamos você a abraçar esta jornada com entusiasmo e curiosidade. A persistência é a chave para a maestria, e com cada linha de código que você escrever, estará mais próximo(a) de transformar suas ideias em aplicativos incríveis.

Bons estudos, e vamos juntos transformar o futuro, com um App de cada vez!

## 1. INTRODUÇÃO À PROGRAMAÇÃO PARA SMARTPHONES

### a. Histórico da programação para *smartphones*

A paisagem tecnológica tem testemunhado mudanças dramáticas nas últimas décadas. No epicentro dessas mudanças, encontra-se o *smartphone*, um dispositivo que se tornou uma extensão inseparável de nossa existência diária. Para entender essa revolução, é imperativo rastrear suas origens e examinar as evoluções que moldaram a programação para *smartphones*.

#### Início com PDAs e Smartphones Primitivos

Antes do advento dos *smartphones* como os conhecemos hoje, o cenário era dominado por PDAs - Personal Digital Assistants ou Assistentes Pessoais Digitais. Estes dispositivos, exemplificados pelo *Palm Pilot* e o *Compaq iPAQ*, focavam principalmente em funções organizacionais como calendário, lista de tarefas e gerenciamento de contatos (West & Mace, 2007). No entanto, com o desejo crescente de conectividade, esses dispositivos logo começaram a incorporar funcionalidades de telefonia, evoluindo para os primeiros *smartphones*. A empresa canadense *BlackBerry* (<https://www.blackberry.com/br/pt>), com seu forte apelo empresarial, foi uma marca pioneira que promoveu a integração de e-mail e naveabilidade na Web em um dispositivo portátil (Goggin, 2011).

#### Evolução dos Sistemas Operacionais Móveis

2007 foi, indiscutivelmente, o ano que redefiniu os *smartphones*, marcado pelo lançamento do *iPhone* da *Apple*. Com o iOS - *iPhone Operating System* (atualmente apenas denominado como iOS), a *Apple* não apenas introduziu um *hardware* estelar, mas um ambiente operacional que redefiniu a experiência do usuário com uma *interface* tátil intuitiva (Kahney, 2014).

Logo depois, o AOS - *Android Operating System* (atualmente apenas denominado Android) baseado no núcleo Linux, respaldado pelo *Google*, entrou no cenário como uma alternativa aberta ao iOS, levando a uma ampla adoção entre os fabricantes de dispositivos devido à sua natureza customizável (Amadeo, 2018). Ambos os Sistemas Operacionais (SOs), com suas lojas de aplicativos associadas, *Google Play* <<https://play.google.com/>> para Android e *App Store* <<https://www.apple.com/br/app-store/>> para iOS, revolucionaram a maneira como os aplicativos são distribuídos e consumidos.

## Desenvolvimento de Apps Nativos e Híbridos

Em resposta ao crescimento explosivo dos SOs móveis, o desenvolvimento de aplicativos também evoluiu. Aplicativos nativos, desenvolvidos para um SO específico usando linguagens como *Swift* para iOS e *Java/Kotlin* para Android, tornaram-se o padrão de oferecendo desempenho otimizado e uma experiência de usuário sem costura (Scharff & Verma, 2011).

No entanto, para atender à demanda de presença em ambas as plataformas sem duplicar o esforço, surgiram *frameworks* para desenvolvimento híbrido. Tecnologias como *React Native* e *Flutter* permitiram aos desenvolvedores criar aplicativos usando um único código base, que poderia ser executado tanto em iOS quanto em Android (Mednieks et al., 2015).

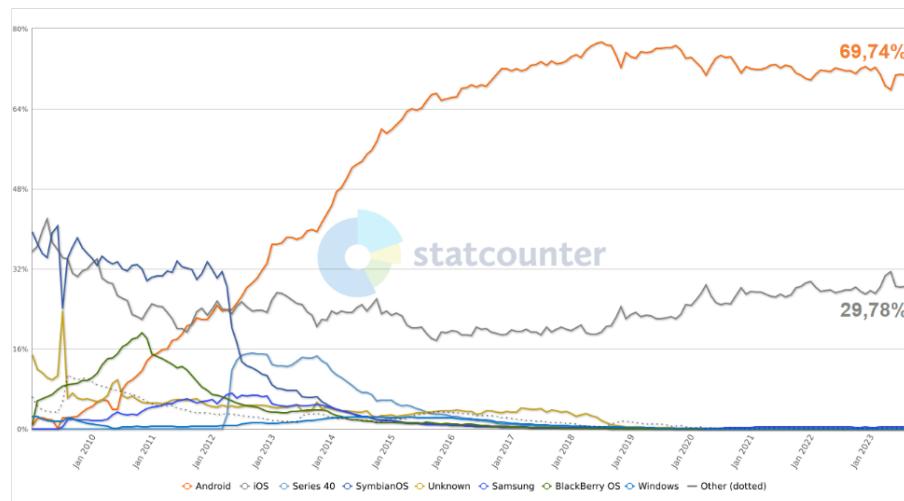
## Principais Plataformas Móveis

Muitas plataformas surgiram na era *touch*, como *Android*, *iOS*, *Windows Phone*, *Bada*, *Web OS*, *Tizen*, *Firefox OS*. Porém apenas sistemas Android e iOS possuem atualmente quantidade relevante no mercado de *smartphones* e isso já vem ocorrendo há bastante tempo. Segundo pesquisa da consultoria americana *Gartner Inc.*, já em 2017 as plataformas Android e iOS dominavam o mercado de *smartphones*, estando presentes em 99,9% dos dispositivos vendidos conforme pode ser visualizado na Tabela 1 a seguir.

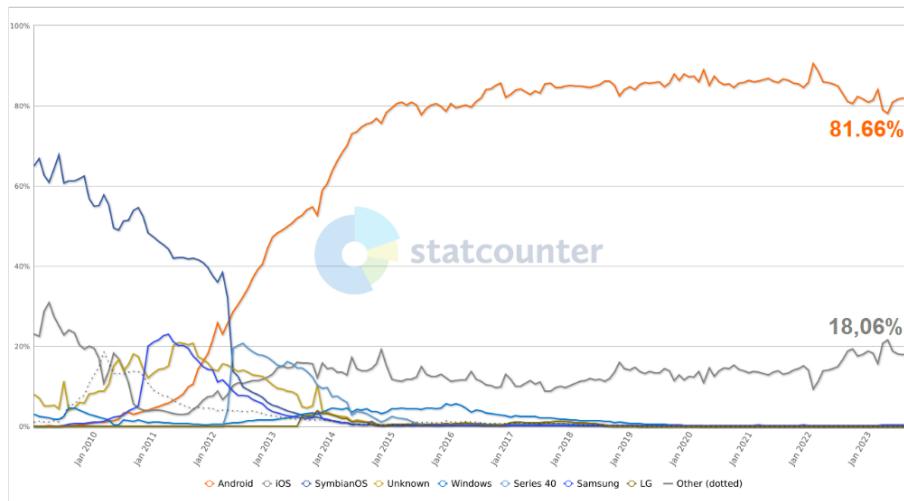
**Tabela 1:** Principais plataformas móveis no mercado em 2017 (Gartner, 2018)

Plataforma	2017 (Unidades)	2017 (% de Mercado)	2016 (Unidades)	2016 (% de Mercado)
<b>Android</b>	1.320.118,1	85,9%	1.268.562,7	84,8%
<b>iOS</b>	214.924,4	14,0%	216.064,0	14,4%
<b>Outros SOs</b>	1.493,0	0,1%	11.332,2	0,8%
<b>Totais</b>	1.536.535,5	100%	1.495.959,0	100%

Trouxemos dados históricos importantes, mas isso é passado! Será? Sempre que vamos nos empenhar em aprender algo visando melhorarmos nossa capacidade profissional e aumentar nossas chances de empregabilidade e/ou acesso de carreira, é importante analisarmos o contexto atual sobre aonde devemos dedicar nossos esforços. Neste sentido trazemos nas Figuras 1 e 2 a seguir, daí que demonstram que não só vale a pena investir seu tempo no contexto mobile, como aonde o mesmo se mostra mais promissor, ou como está o mercado quanto ao uso das plataformas Android e iOS atualmente, no mundo e no Brasil.



**Figura 1:** Distribuição das principais plataformas mobile no mundo entre Jan-2009 e Set-2023 (Statcounter, 2023a)



**Figura 2:** Distribuição das principais plataformas mobile no Brasil entre Jan-2009 e Set-2023 (Statcounter, 2023b)

Agora que já identificamos os números das duas principais plataformas móveis com as quais o mercado vem trabalhando e trabalhando (Mundo: iOS = 29,78% - Android = 69,74% / Brasil: iOS = 18,06% - Android = 81,66%), precisamos escolher com quais ferramentas desenvolveremos os nossos aplicativos. Para tal, é preciso conhecer algumas das opções mais relevantes disponíveis. Na próxima apresentaremos então uma visão geral dos principais ambientes de desenvolvimento para Android e iOS, mas por enquanto vamos conhecer mais sobre as principais características de cada um deles.

#### Você Sabia?

Steven Paul Jobs (1955-2011) é considerado um gênio da área de tecnologia por ter inovado nos setores de computação, animação no cinema, música, telefonia celular, tablets, varejo e publicação digital. Com a criação do *iPhone*, revolucionou o mundo dos *smartphones*, mudando totalmente a forma como as pessoas interagiam com os dispositivos móveis.

Para aqueles interessados em aprofundar sua compreensão sobre a vida e as contribuições de Steve Jobs, recomenda-se assistir aos filmes "Jobs" (2013), estrelado pelo ator Ashton Kutcher, e "Steve Jobs" (2015), com Michael Fassbender no papel principal. Ambos os filmes oferecem vislumbres - embora com abordagens muito diferentes - da personalidade complexa de Jobs, suas realizações, seus desafios e sua jornada para criar uma das empresas mais valiosas do mundo.

## b. Diferenças entre aplicativos para iOS e Android

Como já ilustrado, no mundo dos *smartphones* duas plataformas têm se destacado ao longo dos últimos anos: iOS e Android. Ju- essas plataformas compõem a quase totalidade do mercado de Sistemas Operacionais (SOs) móveis. Mas, apesar de suas semelhanças superficiais, elas são fundamentadas por arquiteturas, filosofias e ecossistemas distintos.

### Sistema Operacional e Kernel

**iOS:** Desenvolvido pela Apple, o iOS é o SO utilizado exclusivamente em dispositivos da Apple, como iPhone, iPad e iPod Touch. iOS é baseado no Darwin, um SO derivado do Unix. Isso proporciona ao mesmo uma base estável e segura, que são duas das razões pelas quais a Apple frequentemente destaca a segurança e a estabilidade como características-chave de seus dispositivos (Levin, 2018).

**Android:** Desenvolvido inicialmente pela Android Inc. a qual foi posteriormente adquirida pela Google LLC, é baseado no kernel Linux. Por ser de código aberto, fabricantes de dispositivos têm a liberdade de personalizar o Android de acordo com suas necessidades, levando a uma variedade de versões e interfaces de usuário no mercado (Seifert, 2019).

### Linguagens Nativas e SDKs - Software Development Kits

**iOS:** Para o desenvolvimento no iOS, a Apple fornece o Kit de Desenvolvimento de Software iOS SDK, que até 2013 focava predominantemente na linguagem Objective-C. No entanto, em 2014, a Apple introduziu a Swift, uma linguagem de programação mais moderna e intuitiva, que agora é amplamente adotada pelos desenvolvedores (Finkel, 2020).

**Android:** O Android oferece o Android SDK, kit que permite o desenvolvimento predominantemente em linguagem Java. Contudo, nos últimos anos, a Kotlin, uma linguagem mais concisa e expressiva, foi introduzida e subsequentemente promovida pelo Google como a linguagem oficial de programação para o Android.

linguagem preferencial para o desenvolvimento Android (Lardinois, 2019).

## Distribuição e Ecossistema

**iOS:** A Apple opera um ecossistema fechado. O único lugar onde os usuários podem baixar aplicativos iOS é através da App Store. A empresa aplica um rigoroso processo de revisão para todos os aplicativos submetidos. Esse modelo fechado, enquanto crítica a Apple ser mais restritiva, também é reconhecido por manter um alto padrão de qualidade e segurança (Cuevas, 2021).

**Android:** Contrasta dramaticamente com o modelo da Apple, permitindo a distribuição de aplicativos através de várias lojas, sendo a Google Play Store a mais proeminente. Além disso, os usuários podem instalar aplicativos de fontes desconhecidas, embora isso possa aumentar riscos de segurança e integridade. Essa abordagem aberta promove uma maior diversidade, mas também apresenta desafios em termos de fragmentação e consistência (Amadeo, 2020).

## Ambientes de Desenvolvimento (IDEs)

Para o desenvolvimento de aplicativos móveis, podemos escolher entre vários IDEs - *Integrated Development Environments* ou Ambientes Integrados de Desenvolvimento populares, dependendo da plataforma que desejarmos segmentar (*Android*, *iOS* ou ambos) e da linguagem de programação que planejamos usar. No quadro *HOW-TO* a seguir, apresentamos alguns IDEs populares para o desenvolvimento móvel, nele você encontrará os links para páginas oficiais dos mesmos. Lembre-se, conhecer diversas plataformas e linguagens é essencial, pois cada uma poderá oferecer recursos específicos para a *Android* ou *iOS*, bem como há também os que atendem ambos os sistemas.

### HOW-TO - Ambientes Integrados de Desenvolvimento

#### Android Studio (Android)

O *Android Studio* é o IDE oficial para desenvolvimento de aplicativos Android. Ele oferece suporte completo para a linguagem *Kotlin* e também para Java. O IDE inclui emuladores Android para teste e depuração.>

<<https://developer.android.com/studio>>

#### Kotlin (Android)

O *Kotlin* é uma linguagem de programação oficialmente suportada para o desenvolvimento de aplicativos Android. Você pode usá-la no *Android Studio* para criar aplicativos *Android* modernos e eficientes.

<<https://kotlinlang.org/>>

#### Xcode (iOS)

Se você está desenvolvendo aplicativos exclusivamente para dispositivos Apple, o *Xcode* é a escolha padrão. Ele suporta *Swift* e *Objective-C* como linguagens de programação e inclui simuladores *iOS* para testes.

<<https://developer.apple.com/xcode/>>

#### AppCode (iOS)

Se você estiver desenvolvendo exclusivamente para *iOS* e preferir uma alternativa ao *Xcode*, o *AppCode* é uma outra opção popular que oferece suporte a *Swift* e *Objective-C*.

<<https://www.jetbrains.com/objc/>>

#### Visual Studio (Multiplataforma)

O *Visual Studio* da Microsoft é um IDE que suporta o desenvolvimento multiplataforma com o uso de ferramentas como *Xamarin* e *Flutter*. Você pode criar aplicativos para *Android*, *iOS* e *Windows* a partir de uma única base de código.

<<https://visualstudio.microsoft.com/pt-br/>>

#### Flutter (Multiplataforma)

Se você deseja criar aplicativos multiplataforma usando a linguagem *Dart*, o *Flutter* oferece uma excelente solução. Você pode usar o *Visual Studio Code* ou o *Android Studio* com o *plug-in Flutter* para desenvolver aplicativos.

<<https://flutter.dev/>>

#### React Native (Multiplataforma)

Para desenvolvimento multiplataforma usando *JavaScript* ou *TypeScript*, o *React Native* é uma opção popular. Você pode usar qualquer IDE de sua escolha, como *Visual Studio Code* ou até mesmo o *Atom*, juntamente com o *React Native CLI*.

<<https://reactnative.dev/>>

#### PhoneGap/Cordova (Multiplataforma)

O *Apache Cordova*, anteriormente conhecido como *PhoneGap*, permite que você crie aplicativos usando *HTML*, *CSS* e *JavaScript*. Você pode usar qualquer IDE que seja compatível com *HTML/JS* para desenvolver aplicativos *Cordova*.

### Python com Kivy (Multiplataforma)

Para criar aplicativos móveis multiplataforma com a linguagem *Python*, você pode usar o *framework Kivy*. Ele permite que você desenvolva aplicativos para *Android*, *iOS* e outras plataformas com uma única base de código em *Python*. Você pode escolher um IDE de sua preferência, como o *Visual Studio Code*, *PyCharm* ou outros, para desenvolver aplicativos com *Kivy*.

<<https://kivy.org/>>

A escolha do IDE dependerá do seu SO, linguagem de programação preferida e plataforma alvo. Certifique-se de escolher um IDE que atenda às suas necessidades e que seja mais adequado para o tipo de aplicativo que você pretende desenvolver, pois isso não só facilitará o seu trabalho, com também aumentará o seu desempenho.

#### Você Sabia?

Como vimos existem várias opções de IDE no mercado, cada uma com suas características. Visando facilitar a sua busca e possível escolha, sugerimos a você que assista aos vídeos a seguir, os quais estão disponíveis no *YouTube* e que certamente não só lhe ensinarão, como também lhe pouparão bastante tempo.

#### As melhores tecnologias para desenvolvimento Mobile | Aquino Explica:

<<https://www.youtube.com/watch?v=DaSJ7tIGnQs>>

#### A forma certa de criar um projeto Flutter pelo VSCode:

<[https://www.youtube.com/watch?v=AI\\_QZ-LEh1I](https://www.youtube.com/watch?v=AI_QZ-LEh1I)>

#### Android Studio ou VS CODE: qual usar para desenvolver aplicativos mobile?

<<https://www.youtube.com/watch?v=2Lp4KqXNJLg>>

## Conhecendo as Linguagens de Programação para Smartphones

Entender as linguagens de programação utilizadas no desenvolvimento de aplicativos para *smartphones* e dispositivos móveis é geral, é fundamental para qualquer desenvolvedor. Aqui estão algumas das linguagens mais comuns e suas respectivas plataformas

### Java (Android)

Java é a linguagem de programação tradicionalmente usada para desenvolver aplicativos *Android*. Ela é suportada pelo *Android Studio* e pelo *Android SDK*. No entanto, o *Kotlin* se tornou a linguagem preferencial para desenvolvimento *Android*, embora o Java a seja amplamente usado.

### Kotlin (Android)

*Kotlin* é uma linguagem de programação moderna e oficialmente suportada pelo *Google* para desenvolvimento *Android*. Ela é considerada mais segura, concisa e expressiva que o *Java*, e muitos desenvolvedores migraram para o *Kotlin* para criar aplicativos *Android*.

### Swift (iOS)

*Swift* é a linguagem de programação oficial da *Apple* para desenvolvimento *iOS*, *macOS*, *watchOS* e *tvOS*. Ela é altamente eficiente e foi projetada para ser fácil de aprender e usar. O *Xcode* é o principal IDE para o desenvolvimento em *Swift*.

### Objective-C (iOS)

Embora a *Swift* tenha se tornado a linguagem de programação preferida para o desenvolvimento *iOS*, o *Objective-C* ainda é usado para aplicativos mais antigos e em algumas bibliotecas e frameworks legados.

### Dart (Flutter - Multiplataforma)

A *Dart* é a linguagem de programação usada pelo *framework Flutter*, que permite o desenvolvimento de aplicativos multiplataforma para *Android*, *iOS* e outros sistemas a partir de uma única base de código. O *Flutter* é uma das mais atrativas alternativas para desenvolvimento móvel.

### **JavaScript/TypeScript (React Native - Multiplataforma)**

A React Native permite que você crie aplicativos móveis para *Android* e *iOS* usando *JavaScript* ou *TypeScript*. Ela usa o mesmo código-fonte para ambas as plataformas, proporcionando um desenvolvimento mais rápido.

### **HTML, CSS e JavaScript (ApacheCordova/PhoneGap - Multiplataforma)**

O Apache Cordova (anteriormente conhecido como PhoneGap) permite que você crie aplicativos móveis usando tecnologias Web padrão, como HTML, CSS e *JavaScript*. Esses aplicativos são empacotados em [contêineres](#)<sup>1</sup> nativos para diferentes plataformas.

### **Python (Kivy - Multiplataforma)**

A linguagem *Python* é usada em conjunto com o framework *Kivy* para criar aplicativos móveis multiplataforma para *Android*, *iOS* e outras plataformas. O *Kivy* permite que você desenvolva aplicativos com uma única base de código em *Python*.

### **C/C++ (Android e iOS)**

Embora Java, *Kotlin* e *Swift* sejam as linguagens de programação preferenciais para o desenvolvimento *Android* e *iOS*, respectivamente, você também pode usar C ou C++ para criar bibliotecas nativas e integrá-las em seus aplicativos móveis. Essa abordagem é usada principalmente para desempenho crítico e aplicativos de alto desempenho.

### **Lua (Corona SDK - Multiplataforma)**

O Corona SDK usa a linguagem *Lua* para desenvolver aplicativos multiplataforma. *Lua* é uma linguagem de *script* leve e fácil de aprender.

### **JavaScript (NativeScript - Multiplataforma)**

O NativeScript permite que você desenvolva aplicativos móveis nativos para *Android* e *iOS* usando *JavaScript* ou *TypeScript*. É uma alternativa ao React Native e ao desenvolvimento puramente nativo.

### **Ruby (RubyMotion - iOS e Android)**

*RubyMotion* permite que você desenvolva aplicativos para *iOS* e *Android* usando *Ruby* como linguagem de programação. É uma opção para desenvolvedores que já estão familiarizados com *Ruby*.

### **Go (Gomobile - Android e iOS)**

O Gomobile é uma ferramenta que permite usar a linguagem de programação Go para criar bibliotecas nativas que podem ser usadas em aplicativos *Android* e *iOS*. É mais comumente usado para aplicativos de alto desempenho.

### **Rust (Rustfor Android/iOS)**

Rust é uma linguagem de programação de sistema que pode ser usada para desenvolvimento móvel por meio de [bindings](#)<sup>2</sup> e ferramentas específicas, sendo reconhecida por sua segurança e desempenho.

Embora essas linguagens possam ser usadas no desenvolvimento móvel, é importante notar que a escolha da linguagem geralmente está relacionada à plataforma alvo, às diretrizes da plataforma e às bibliotecas/frameworks disponíveis. A seleção da linguagem também depende das necessidades específicas do projeto e da experiência do desenvolvedor e/ou da equipe de desenvolvimento. Portanto, a escolha da linguagem deve ser feita considerando diversos fatores.

Cada linguagem tem suas vantagens e desvantagens, e a escolha depende das necessidades do seu projeto, da plataforma alvo e da sua preferência pessoal. É importante considerar ainda, as diretrizes e ferramentas oferecidas pelas empresas responsáveis pelas plataformas, como por exemplo o Google e a Apple, ao escolher uma linguagem de programação para desenvolvimento móvel.

<sup>1</sup> São pacotes leves do código do aplicativo com dependências, como versões específicas de ambientes de execução de linguagem de programação e bibliotecas necessárias para executar seus serviços software.

<sup>2</sup> Bindings são estruturas programadas para otimizar ações ou conjuntos de ações “teclas de atalho” para serem executadas diretamente através da tecla de funções pelo prompt de comandos.

## **2. FUNDAMENTOS DA PROGRAMAÇÃO PARA SMARTPHONES**

Os fundamentos da programação para *smartphones* são os princípios básicos que você precisa entender e dominar para criar aplicativos móveis de forma eficaz. Aqui estão alguns dos principais fundamentos da programação para tal objetivo.

### Linguagens de Programação

Primeiro, você precisa escolher uma linguagem de programação apropriada para o SO alvo. Por exemplo, *Kotlin* e *Java* são comuns para o desenvolvimento Android, enquanto *Swift* e *Objective-C* são usados para iOS. Linguagens como *JavaScript* (com *React Native* ou *NativeScript*) e *Dart* (com *Flutter*) permitem o desenvolvimento multiplataforma.

### IDE - Integrated Development Environment ou Ambiente de Desenvolvimento Integrado

Familiarize-se com o IDE usado para desenvolvimento na plataforma escolhida. *Android Studio* é amplamente usado para Android e *Xcode* é o principal IDE para iOS. Outros IDEs, como *Visual Studio Code*, podem ser usados com certos frameworks multiplataforma.

### UI/UX Design

Compreenda os princípios de *design de interface* do usuário (UI - *User Interface*) e experiência do usuário (UX - *User eXperience*). A aparência e a usabilidade de um aplicativo móvel são cruciais para o sucesso do mesmo. Você precisará aprender a criar layouts, usar componentes de interface do usuário e lidar com a navegação.

### Gerenciamento de Estado

Muitos aplicativos móveis precisam gerenciar o estado da aplicação. Isso inclui lidar com dados, configurações do usuário e o ciclo de vida da aplicação. Aprenda a usar estruturas de gerenciamento de estado, como o *ViewModel* no Android ou o *AppState* no iOS.

### Acesso a Dados

Saiba como acessar e manipular dados em seu aplicativo, seja por meio de APIs - *Application Programming Interface*<sup>1</sup> ou Interfaces de Programação de Aplicação da Web, bancos de dados locais ou armazenamento em nuvem. Isso envolve o uso de solicitações HTTP/HTTPS, consultas a bancos de dados e armazenamento de dados em cache.

### Integração de Hardware

Os *smartphones* têm sensores e recursos específicos, como câmeras, GPS, acelerômetros, giroscópio, magnetômetro, biométrico, luz do ambiente, proximidade, etc. Aprenda a acessar e usar esses recursos em seu aplicativo quando necessário, pois isso poderá fornecer excelentes diferenciais ao mesmo.

### Testes e Depuração

Desenvolver aplicativos móveis requer testes rigorosos em dispositivos reais, emuladores e/ou simuladores. Saiba como depurar problemas e usar ferramentas de teste, como depuradores e emuladores.

### Segurança

Compreenda as práticas recomendadas de segurança para proteger os dados dos usuários e a integridade do seu aplicativo. Isso inclui criptografia, autenticação de usuário e autorização entre outras possibilidades.

### Publicação nas Lojas de Aplicativos

Saiba como preparar seu aplicativo para a publicação nas lojas de aplicativos (*Google Play Store* para Android e *App Store* para iOS). Isso inclui a geração de recursos gráficos, a configuração de listagens de aplicativos e a conformidade com as diretrizes da(s) loja(s).

### Atualizações e Manutenção

Entenda a importância de manter seu aplicativo móvel com atualizações regulares e funcionando em diferentes versões do SO native. Isso envolve testes, correções de bugs, melhorias de desempenho e adição de novos recursos.

### Documentação e Comunicação

Documente seu código e comunique-se com a comunidade de desenvolvedores, colegas de equipe e usuários. A colaboração e a documentação são fundamentais para um desenvolvimento e manutenção eficazes.

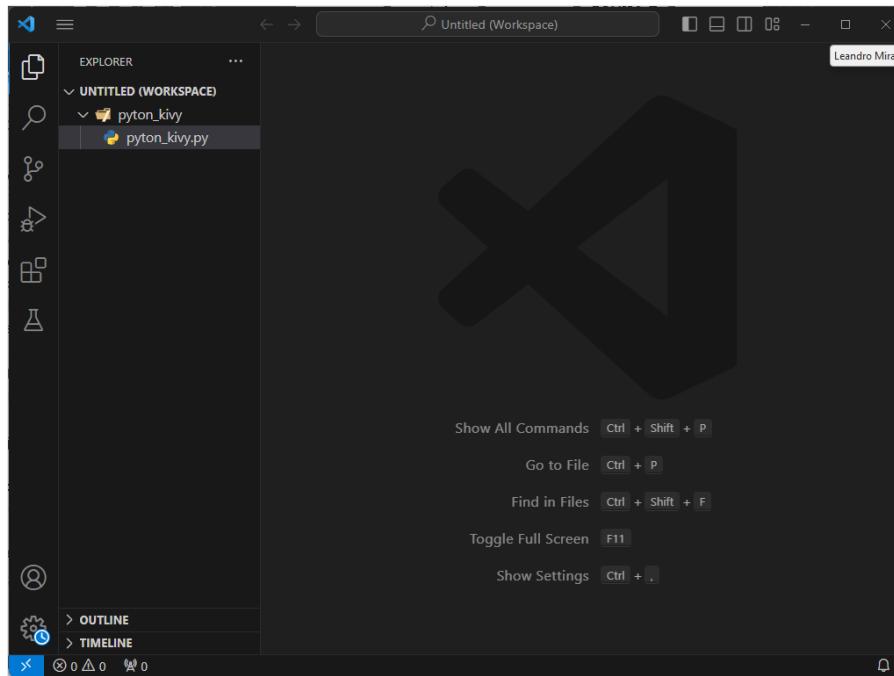
Dominar esses fundamentos é essencial para se tornar um desenvolvedor de aplicativos móveis competente. À medida que você ganha experiência, também pode explorar tópicos mais avançados, como o desenvolvimento de Jogos, Realidade Aumentada (AR - Augmented Reality) e Inteligência Artificial (AI - Artificial Intelligence) em aplicativos móveis.

### Colocando a mão na massa

Como já esclarecemos, para desenvolvermos aplicativos móveis temos que fazer escolhas quanto à(s) plataforma(s), o IDE e a linguagem de programação. Em nosso curso nos focaremos na plataforma Android, utilizando o aplicativo *Visual Studio Code* como ferramenta de desenvolvimento de aplicação *mobile*, conjuntamente com o *framework Kivy* e com a codificação ocorrendo em linguagem *Python*.

### **Visual Studio Code (VSCode) com Kivy**

O VSCode <<https://code.visualstudio.com/download>> é um editor de código fonte gratuito e de código aberto desenvolvido pela *Microsoft*, o mesmo se destaca por sua flexibilidade e pela ampla gama de extensões disponíveis, suportando uma variedade de linguagens de programação e possuindo ainda, suporte integrado para *JavaScript*, *TypeScript* e *Node.js*. A Figura 3 a seguir, ilustra a tela inicial do *VSCode*.



**Figura 3:** Tela inicial do *Visual Studio Code*

### **Para Aprender e Treinar**

Para iniciarmos com a implementação e testes de nossos exemplos, você deverá instalar e/ou já ter instalado um IDE em seu computador. Em nosso curso optamos por utilizar o *VSCode* da empresa *Microsoft*, o qual apesar de possuir um processo bastante simples de instalação e configuração, por questões de foco e tempo não nos ateremos em descrever. Deste modo, sugerimos que você assista com atenção ao vídeo intitulado “**Instalação do VSCode para Programação em JavaScript e Python**”, disponível no *YouTube*, o qual explica rápida e eficientemente como tal processo deve ocorrer.

<sup>1</sup> APIs são essencialmente, conjuntos de regras e protocolos que permitem que diferentes softwares interajam entre si.

## **Fundamentos da programação para smartphones usando Python com o framework Kivy**

### **a. Variáveis, Tipos de Dados e Operadores**

#### **Variáveis**

*Python* é uma linguagem de programação de alto nível, com estruturas de dados integradas também de alto nível e vinculação dinâmica. É interpretada e orientada a objetos, ela diferencia-se de outras linguagens de programação por sua sintaxe fácil de escrever e compreender, o que a torna atraente tanto para iniciantes quanto para profissionais experientes. A vasta aplicabilidade e o suporte de bibliotecas *Python* permitem a criação de softwares e produtos altamente versáteis e escaláveis com base no mundo real.

Em *Python* (e também com o *Kivy*), as variáveis são usadas para armazenar dados. Você pode pensar nelas como recipientes que guardam informações que você deseja usar em seu programa. Para criar uma variável em *Python*, basta atribuir um valor a um nome.

Exemplo de atribuição de valores às variáveis **Nome** e **Idade** em código *Python*:

**Nome = "João"**

**Idade = 30**

Em nosso exemplo inicial, "**nome**" e "**idade**" são variáveis que armazenam uma *string* e um número inteiro, respectivamente.

## Tipos de Dados

*Python* possui vários tipos de dados comuns, dentre os quais destacamos:

- **Inteiro (int)**: Armazena números inteiros.
  - Exemplo: **idade = 30**
- **Flutuante (float)**: Armazena números decimais.
  - Exemplo: **altura = 1.75**
- **String (str)**: Armazena texto.
  - Exemplo: **nome = "Maria"**
- **Booleano (bool)**: Armazena valores booleanos/lógicos verdadeiros ou falsos (True ou False).
  - Exemplo: **tem\_carteira = True**
- **Lista (list)**: Armazena uma coleção ordenada de itens.
  - Exemplo: **frutas = ["maçã", "banana", "laranja"]**
- **Dicionário (dict)**: Armazena pares chave-valor.
  - Exemplo: **pessoa = {"nome": "Carlos", "idade": 25}**
- **Tupla (tuple)**: Semelhante a uma lista, mas imutável.
  - Exemplo: **coordenadas = (3, 4)**
- **Conjunto (set)**: Armazena itens únicos sem ordem específica.
  - Exemplo: **numeros = {1, 2, 3}**

## Operadores

Operadores são usados para realizar operações em variáveis e valores. Alguns dos operadores mais comuns incluem:

- **Operadores Aritméticos**: Usados para realizar operações matemáticas.
  - + (adição), - (subtração), \* (multiplicação), / (divisão), % (módulo para resto da divisão), \*\* (potência).
- **Operadores de Comparaçāo**: Usados para comparar valores.
  - == (igual a), != (diferente de), < (menor que), > (maior que), <= (menor ou igual a), >= (maior ou igual a).
- **Operadores Lógicos**: Usados para combinar expressões booleanas.
  - **and** (e lógico), **or** (ou lógico), **not** (não lógico).

A Figura 4, ilustra um exemplo de uso de variáveis, tipos de dados e operadores em *Python* com *Kivy*.

```

1 # Variáveis
2 nome = "Maria"
3 idade = 25
4
5 # Tipos de Dados
6 altura = 1.75
7 tem_carteira = True
8
9 # Operadores
10 soma = idade + 5
11 eh_adulto = idade >= 18
12 frutas = ["maçã", "banana"]
13 tem_morango = "morango" in frutas
14

```

**Figura 4:** Exemplo do uso de variáveis em Python

### b. Estruturas de Controle de Fluxo (Condicionais e Repetições)

Além de variáveis, tipos de dados e operadores, as estruturas de controle de fluxo são fundamentais para a programação em Python. Elas permitem que você tome decisões e repita ações com base em condições. Vamos explorar as estruturas de controle de fluxo.

#### Estruturas Condicionais (if, elif, else)

As estruturas condicionais permitem que você tome decisões com base em condições. O **if** verifica se uma condição é verdadeira e executa um bloco de código se a mesma for verdadeira. O **elif** (abreviação de "else if") é usado para verificar condições adicionais, e o **else** é executado quando nenhuma das condições anteriores é verdadeira. A Figura 5, exibe um exemplo de tal estrutura.

```

idade = 25

if idade < 18:
    print("Menor de idade")
elif idade >= 18 and idade < 65:
    print("Adulto")
else:
    print("Idoso")

```

**Figura 5:** Exemplo de estrutura condicional em Python

#### Estruturas de Repetição (For e While)

As estruturas de repetição permitem que você execute um bloco de código várias vezes. O **for** é usado para iterar (repetir) sobre uma sequência (como uma lista) e executar um bloco de código para cada item na sequência. O **while** executa um bloco de código enquanto uma condição é verdadeira. As Figuras 6 e 7 a seguir, ilustram exemplos de ambas as estruturas.

Usando **for** para iterar em uma lista.

```

frutas = ["maçã", "banana", "laranja"]

for fruta in frutas:
    print("Gosto de", fruta)

```

**Figura 6:** Exemplo de uso do **for** para iterar lista em Python

Usando **while** para contar até 5.

```

contador = 1

while contador <= 5:
    print("Contagem:", contador)
    contador += 1

```

Figura 7: Exemplo do uso da estrutura condicional `while` em Python

### Para Aprender e Treinar

O Kivy <<https://kivy.org/>> é uma biblioteca *Open Source* escrita em *Python* para o desenvolvimento de aplicações multiplataforma seja para ambientes *desktop* (*Windows*, *Linux* e *MacOS*) ou para ambientes móveis (*Android* e *iOS*). Lançado em 2011 e com várias versões de atualização, desde então, o Kivy é uma ótima alternativa quando queremos desenvolver uma aplicação nativa para qualquer plataforma.

O framework Kivy não faz parte dos pacotes *Python* e/ou *VSCode*, portanto será necessário que você o instale em seu computador para poder usufruir de suas potencialidades. Como já mencionamos em relação ao *VSCode* e faremos novamente à frente, em nosso curso não possuímos tempo hábil para abordar os detalhes de instalação e configuração dos diferentes softwares que utilizaremos, sugerimos então que você assista com atenção ao vídeo intitulado “**Tutorial de instalação do framework Kivy no VSCode**”, disponível no *YouTube*, o qual certamente ajudará a compreender tais detalhes e ainda irá economizar bastante de seu tempo.

Essas estruturas de controle de fluxo são essenciais para criar aplicativos interativos. Por exemplo, você pode usá-las para tomar decisões com base nas ações do usuário ou para iterar sobre elementos da interface do usuário em um aplicativo móvel Kivy.

A Figura 8, ilustra um exemplo simples que combina variáveis, estruturas condicionais e de repetição em um aplicativo móvel Kivy.

```

from kivy.app import App
from kivy.uix.button import Button

class MeuApp(App):
    def build(self):
        layout = Button(text="Clique-me!")
        layout.bind(on_press=self.on_button_click)
        return layout

    def on_button_click(self, instance):
        idade = 25

        if idade < 18:
            mensagem = "Menor de idade"
        elif idade >= 18 and idade < 65:
            mensagem = "Adulto"
        else:
            mensagem = "Idoso"

        instance.text = mensagem

if __name__ == "__main__":
    MeuApp().run()

```

Figura 8: Exemplo combinando variáveis, estruturas condicionais e de repetição em um aplicativo móvel Kivy

No exemplo, um botão exibe uma mensagem com base na idade quando é clicado. Isso demonstra como variáveis, estruturas condicionais e eventos podem ser usados em conjunto em um aplicativo *Kivy*.

### c. Funções e Métodos

Compreender funções e métodos é fundamental para a programação em *Python*, pois o seu uso permite que você modularize seu código, torne-o mais legível e reutilize blocos de código. Não se preocupe, por iremos explorar esses conceitos mais à frente.

#### Funções em Python

Uma função em *Python* é um bloco de código que executa uma tarefa específica. Ela pode receber argumentos (dados de entrada) para realizar operações com esses argumentos e retornar um resultado. A Figura 9 a seguir, ilustra um exemplo de uma função simples que calcula a soma de dois números.

```
def somar(a, b):
    resultado = a + b
    return resultado
```

Figura 9: Exemplo de função em *Python*

Neste exemplo, a função **somar** recebe dois argumentos **a** e **b**, calcula a soma deles e retorna o **resultado**.

#### Chamando uma Função

Para chamar uma função em *Python*, você simplesmente a invoca (chama) pelo nome e passa os argumentos necessários. A Figura 10 a seguir, ilustra como exemplo a chamada de função **somar** com a passagem dos argumentos **5** para **A** e **3** para **B**.

```
resultado_soma = somar(5, 3)
print(resultado_soma) # Isso imprimirá 8
```

Figura 10: Exemplo de chamada de função em *Python*

#### Métodos em Python

Os métodos são funções que estão associadas a objetos. Em *Python*, quase tudo é um objeto, e muitos objetos têm métodos que pode chamar para realizar operações específicas. Por exemplo, listas têm métodos como **append()**, **remove()**, e strings têm métodos como **split()** e **join()**. Você chama os métodos usando a notação de ponto (**objeto.metodo()**). A Figura 11 a seguir, ilustra um exemplo de uso de métodos em uma lista.

```
frutas = ["maçã", "banana", "laranja"]
frutas.append("uva") # Adiciona "uva" à lista
frutas.remove("banana") # Remove "banana" da lista
```

Figura 11: Exemplo de listas com a chamada dos métodos **append()** e **remove()**

#### Funções em Métodos *Kivy*

Ao desenvolver aplicativos móveis com *Kivy*, você criará funções personalizadas para lidar com eventos e comportamentos específicos de *widgets*<sup>1</sup>. Por exemplo, você pode definir uma função para tratar o evento de clique de um botão. A Figura 12 a seguir, ilustra um exemplo de tratamento de evento.

```

from kivy.app import App
from kivy.uix.button import Button

class MeuApp(App):
    def build(self):
        botao = Button(text="Clique-me!")
        botao.bind(on_press=self.on_button_click)
        return botao

    def on_button_click(self, instance):
        instance.text = "Clicou!"

```

**Figura 12:** Função para tratar o evento de clique de um botão

Neste exemplo, `on_button_click` é uma função personalizada do módulo `kivy.uix` a qual é chamada quando o botão é clicado. Ela altera o texto do botão para "Clicou!".

Entender funções e métodos é essencial para a criação de aplicativos móveis bem-organizados e funcionais em *Python* com *Kivy*. Permitirá que você quebre seu código em partes menores e reutilizáveis, facilitando a manutenção e o desenvolvimento de recursos complexos.

<sup>1</sup> São elementos de uma GUI - *Graphical User Interface* ou Interface Gráfica do Usuário, que fazem parte da experiência do usuário. O módulo `kivy.uix` contém classes de criação e gerenciamento de widgets (botões, controles deslizantes, barras de progresso, campos de entrada de texto, botão de interruptor, etc.).

#### d. Programação Orientada a Objetos (POO) aplicada a smartphones

A jornada da programação é, muitas vezes, complexa e sinuosa, mas também é inegavelmente gratificante. Quando se fala em desenvolvimento moderno de software, um conceito que frequentemente emerge como um pilar central é o de Orientação a Objetos. Como Mark Lutz menciona em seu livro *Learning Python* (2013), "*No mundo do Python, a orientação a objetos é um estilo de programação que se concentra na criação de objetos reutilizáveis...*". Entender esse paradigma é crucial para quem deseja elevar suas habilidades de programação a um próximo nível, especialmente em *Python*, uma linguagem que incorpora a OO em sua essência.

A OO não é apenas uma característica da linguagem *Python*; é uma filosofia de *design de software* que se baseia na ideia de que estruturar nosso código em torno de "objetos" - entidades que contêm tanto dados quanto as operações associadas a eles mesmos - podemos criar sistemas mais modulares, escaláveis e gerenciáveis. A OO traz ao mundo da programação uma representação mais próxima de como percebemos e interagimos com o mundo real, e em *Python* ela não é apenas uma característica adicional, mas é o cerne da linguagem, influenciando até mesmo suas estruturas mais básicas.

Por que, então, a OO é tão enfatizada em *Python*? Uma das principais razões é a flexibilidade e a clareza que ela oferece. Em *Python*, tudo é um objeto, seja um número, uma *string* ou até mesmo funções. Esse *design* intrinsecamente orientado a objetos permite aos desenvolvedores pensarem em problemas de uma maneira mais intuitiva, organizando o código em torno de entidades e suas interações. Com essa abordagem, é mais fácil modelar problemas complexos, como ressalta (Slatkin, 2019).

À medida que avançarmos nesta seção, desvendaremos os fundamentos da OO em *Python*, explorando os conceitos-chave, como classes, objetos, herança, polimorfismo e encapsulamento. Através de exemplos práticos e analogias, pretendemos tornar essa importante área da programação acessível a você, independentemente de sua experiência prévia.

Como Alan Kay, um dos pioneiros da OO, afirmou:

"A perspectiva orientada a objetos é simplesmente a paralela que a maneira mais complexa de sistemas funcionarem bem com que eles sejam mais como comunidades simples e como máquinas complicadas". (Kay)

Portanto, embarquemos juntos nesta aventura e vamos descobrir juntos a simplicidade e o poder da orientação a objetos em *Python* para o desenvolvimento de software moderno.

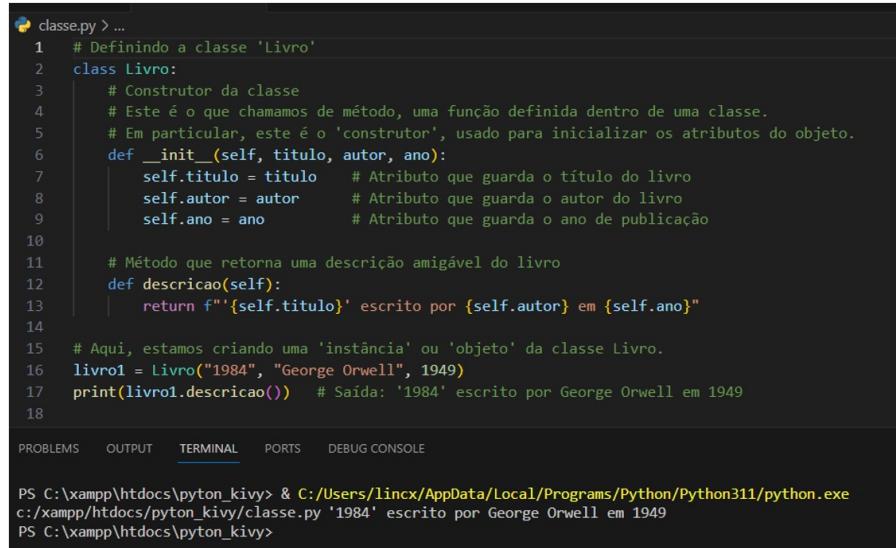
## Conceitos Chave da POO

### Classes e Objetos

A linguagem de programação *Python* é conhecida por sua simplicidade e elegância, tornando-a acessível para novatos e poderoso para os especialistas. No centro dessa simplicidade está o conceito de Orientação a Objetos (OO), e as entidades cruciais dessa abordagem são as classes e os objetos. As classes são fundamentos da programação OO e atuam como moldes para criar objetos (Matthes, 2019).

Então, o que exatamente são classes e objetos? Imagine uma classe como sendo um projeto ou um modelo. Por exemplo, ao planejar a construção de uma casa, arquitetos usam plantas. Essas plantas (as classes) não são casas reais, mas guias sobre como uma casa deve ser construída. Já os objetos são as instâncias reais criadas a partir desses projetos. Portanto, seguindo nossa analogia, a casa construída com base na planta, é um objeto.

Ilustrando isso com um exemplo prático, vamos considerar a ideia de um livro em uma biblioteca. Podemos ter uma classe *Livro* que define as características gerais de um livro, mas não especifica qual livro em particular. A Figura 13 ilustra a construção da classe *Livro*.



```

classe.py > ...
1 # Definindo a classe 'Livro'
2 class Livro:
3     # Construtor da classe
4     # Este é o que chamamos de método, uma função definida dentro de uma classe.
5     # Em particular, este é o 'construtor', usado para inicializar os atributos do objeto.
6     def __init__(self, titulo, autor, ano):
7         self.titulo = titulo    # Atributo que guarda o título do livro
8         self.autor = autor      # Atributo que guarda o autor do livro
9         self.ano = ano          # Atributo que guarda o ano de publicação
10
11    # Método que retorna uma descrição amigável do livro
12    def descricao(self):
13        return f'{self.titulo}' escrito por {self.autor} em {self.ano}'
14
15    # Aqui, estamos criando uma 'instância' ou 'objeto' da classe Livro.
16    livro1 = Livro("1984", "George Orwell", 1949)
17    print(livro1.descricao())  # Saída: '1984' escrito por George Orwell em 1949
18

```

PROBLEMS    OUTPUT    **TERMINAL**    PORTS    DEBUG CONSOLE

```

PS C:\xampp\htdocs\python_kivy> & C:/Users/lincx/AppData/Local/Programs/Python/Python311/python.exe
c:/xampp\htdocs\python_kivy/classe.py '1984' escrito por George Orwell em 1949
PS C:\xampp\htdocs\python_kivy>

```

**Figura 13:** Exemplo de construção de uma Classe

Note que usamos um método especial chamado `__init__`. Este é o "construtor" da classe, que é automaticamente invocado (chama-se sempre que criamos uma nova instância (representação) da classe. Ele serve para inicializar os atributos dos objetos, ou seja, definir valores iniciais para eles.

A beleza da OO reside em sua capacidade de modelar o mundo real de forma intuitiva. À medida que avançamos no estudo de classes e objetos em *Python*, torna-se evidente que este paradigma nos ajuda a organizar nosso pensamento e nosso código de maneira mais alinhada com a realidade (Ramalho, 2015).

## Atributos e Métodos

Mergulhando mais fundo na essência da OO em *Python*, nos deparamos com dois conceitos cruciais: **atributos** e **métodos**. Na vastidão da programação moderna, entender estas facetas é imprescindível para a modelagem eficaz de soluções em software. Seja (Lutz, 2013), as classes encapsulam dados para o programa e fornecem métodos para manipular esses dados, um paradigma que é organização e a clareza do código.

Vamos começar com os **atributos**. Em sua essência, atributos são variáveis associadas a uma classe. Eles representam as características ou propriedades de um objeto. Por exemplo, em um sistema de gerenciamento de bibliotecas, um livro pode ter atributos como título, autor e ano de publicação.

Os **métodos**, por outro lado, são funções associadas a uma classe. Eles definem o comportamento dos objetos. No contexto da biblioteca, um método poderia ser a ação de um livro sendo emprestado ou retornado.

Para entendermos melhor, vejamos o exemplo prático ilustrado na Figura 14.

```

1  classe.py > ...
2
3  class Livro:
4      # O construtor inicializa os atributos do objeto.
5      def __init__(self, titulo, autor, ano):
6          self.titulo = titulo    # Atributo do título
7          self.autor = autor      # Atributo do autor
8          self.ano = ano          # Atributo do ano de publicação
9          self.emprestado = False # Atributo para verificar se o livro está emprestado
10
11     # Método para emprestar o livro
12     def emprestar(self):
13         if not self.emprestado:
14             self.emprestado = True
15             return f'{self.titulo}' agora está emprestado."
16         return f'{self.titulo}' já está emprestado."
17
18     # Método para devolver o livro
19     def devolver(self):
20         if self.emprestado:
21             self.emprestado = False
22             return f'{self.titulo}' foi devolvido."
23         return f'{self.titulo}' não estava emprestado."
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39

```

**Figura 14:** Exemplo da construção de Atributos e Métodos

A Figura 15 ilustra a execução da classe e seus métodos.

```

1  classe.py > ...
2
3
4  # Criando um objeto da classe Livro
5  meu_livro = Livro("Cem Anos de Solidão", "Gabriel García Márquez", 1967)
6
7  # Emprestando o livro
8  print(meu_livro.emprestar())
9  # Saída esperada: 'Cem Anos de Solidão' agora está emprestado.
10
11 # Tentando emprestar novamente
12 print(meu_livro.emprestar())
13 # Saída esperada: 'Cem Anos de Solidão' já está emprestado.
14
15 # Devolvendo o livro
16 print(meu_livro.devolver())
17 # Saída esperada: 'Cem Anos de Solidão' foi devolvido.
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39

```

PROBLEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE

```

PS C:\xampp\htdocs\pyton_kivy> & C:/Users/lincx/AppData/Local/Programs/Python/Python311/python.exe
c:/xampp/htdocs/pyton_kivy/classe.py
'Cem Anos de Solidão' agora está emprestado.
'Cem Anos de Solidão' já está emprestado.
'Cem Anos de Solidão' foi devolvido.
PS C:\xampp\htdocs\pyton_kivy>

```

**Figura 15:** Saída do exemplo da construção de Atributos e Métodos

No código acima, a classe **Livro** possui atributos (como **titulo** e **autor**) e métodos (como **emprestar** e **devolver**). Estes permitem apenas armazenar dados sobre cada livro, mas também interagir com esses dados de maneira significativa.

Com este exemplo simples, é possível perceber a interação dos atributos e métodos, dando vida e funcionalidade aos objetos da classe. O que demonstra de forma prática como a OO em *Python* fornece intrinsecamente ferramentas para modelar o mundo ao redor de forma concisa e intuitiva.

Conforme seguimos nossa jornada de aprendizado, é fundamental entender que atributos e métodos são a espinha dorsal da OO em *Python*. O paradigma de OO proporciona uma estrutura sólida e coerente, permitindo que os programadores modelem e resolvam problemas complexos de uma maneira intuitiva (Phillips, 2018).

## Encapsulamento

Dentre os pilares da POO, o encapsulamento ocupa um espaço de destaque. No cerne do conceito, encontra-se a ideia de restringir o acesso direto a alguns dos componentes de um objeto, mantendo-os escondidos do mundo exterior. Tal restrição não só protege a integridade dos dados, como também promove uma maior organização e coesão dos códigos desenvolvidos.

Em *Python*, o encapsulamento é alcançado por meio de convenções, e não por restrições forçadas como em algumas outras linguagens. Para tal, usamos o prefixo de sublinhado "\_" antes de um atributo ou método para indicar que ele é protegido (por convenção) e que não deve ser acessado diretamente. No entanto, isso não impede tecnicamente o seu acesso. Por outro lado, o uso de dois

sublinhados “\_\_”, por exemplo atributoPrivado (mecanismo chamado *mangling*<sup>1</sup>), torna o atributo praticamente inacessível por normais.

No contexto de nossa biblioteca, suponhamos que desejamos rastrear a quantidade de vezes que um livro é emprestado, mas não queremos que essa contagem possa ser alterada diretamente. A Figura 16, demonstra como o encapsulamento pode ser aplicado para propósito.

```

class.py > ...
1  class Livro:
2      def __init__(self, titulo, autor):
3          self.titulo = titulo      # Atributo público
4          self.autor = autor        # Atributo público
5          self.__emprestimos = 0    # Atributo privado (encapsulado)
6
7      # Método público para emprestar o livro
8      def emprestar(self):
9          self.__emprestimos += 1    # Incrementa o contador de empréstimos
10         return f"'{self.titulo}' foi emprestado {self.__emprestimos} vez(es)."
11
12     # Método para verificar quantas vezes o livro foi emprestado
13     def vezes_emprestado(self):
14         return self.__emprestimos  # Retorna o valor encapsulado
15

```

Figura 16: Exemplo de Encapsulamento

Neste exemplo, \_\_emprestimos é um atributo privado, portanto somente métodos dentro da classe **Livro** podem acessá-lo e modificá-lo diretamente. Isso garante que a contagem de empréstimos seja protegida e gerenciada de forma controlada. A Figura 17, exemplifica esse conceito de forma prática.

```

class.py > ...
15
16
17 # Criando um objeto da classe Livro
18 meu_livro = Livro("Dom Casmurro", "Machado de Assis")
19
20 # Emprestando o livro algumas vezes
21 print(meu_livro.emprestar()) # Saída esperada: 'Dom Casmurro' foi emprestado 1 vez(es).
22 print(meu_livro.emprestar()) # Saída esperada: 'Dom Casmurro' foi emprestado 2 vez(es).
23
24 # Tentando acessar diretamente o atributo __emprestimos (isso deve resultar em um erro)
25 # print(meu_livro.__emprestimos) # Descomente essa linha para verificar que o
26 # acesso direto resulta em erro
27 # Usando o método para verificar quantas vezes o livro foi emprestado
28 print(f"O livro foi emprestado {meu_livro.vezes_emprestado()} vez(es).")
29 # Saída esperada: O livro foi emprestado 2 vez(es).
30
31
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
PS C:\xampp\htdocs\python_kivy & C:/Users/lincx/AppData/Local/Programs/Python/Python311/python.exe
c:/xampp/htdocs/python_kivy/classe.py
'Dom Casmurro' foi emprestado 1 vez(es).
'Dom Casmurro' foi emprestado 2 vez(es).
O livro foi emprestado 2 vez(es).
PS C:\xampp\htdocs\python_kivy>

```

Figura 17: Saída de exemplo de Encapsulamento

A prática do encapsulamento vai além da mera restrição de acesso. Ela orienta o desenvolvedor a pensar mais cuidadosamente a arquitetura do código, promovendo *designs* mais robustos e evitando interferências indesejadas. Goodrich et al. (2019), reforça ainda que encapsular adequadamente os dados e comportamentos de um objeto é fundamental para criar sistemas mais seguros e gerenciáveis.

<sup>1</sup> *mangling* é o nome usado para atributos de classe que não se deseja que subclasses usem.

## Herança

Herança é outro dos pilares fundamentais da POO. Ela permite que novas classes sejam criadas com base em classes existentes herdando seus atributos e comportamentos, tal característica oferece uma maneira eficaz de promover a reutilização de código, ao mesmo tempo que estabelece uma relação natural entre a classe base e a derivada (Lutz, 2019).

Uma das belezas da herança é que ela facilita a representação de relações hierárquicas. Por exemplo, podemos pensar em uma classe "**Animal**" e, a partir dela, derivar outras classes, como "**Mamífero**" ou "**Ave**". Dessa forma, características gerais de animais (comer ou dormir) podem ser definidas na classe "**Animal**", enquanto características específicas (como voar) podem ser definidas na classe "**Ave**".

Em *Python*, a herança é simplesmente implementada através da especificação da classe base entre parênteses na definição da derivada. Durante esse processo, é importante compreender a diferença entre "herança simples" e "herança múltipla". Em *Python*, a herança múltipla é permitida, ou seja, uma classe pode herdar de múltiplas classes base, o que a diferencia de algumas outras linguagens de programação (Van Rossum & Drake, 2011).

Na Figura 18 ilustramos a implementação de herança por meio de nosso exemplo de biblioteca. Suponha que tenhamos uma classe base **ItemDeBiblioteca** e que desejamos criar classes derivadas como **Livro** e **DVD**.

```

class.py > ...
1  # Definindo a classe base ItemDeBiblioteca
2  class ItemDeBiblioteca:
3      # Construtor da classe base
4      def __init__(self, titulo):
5          self.titulo = titulo # Atributo comum a todos os itens da biblioteca
6
7      # Método para exibir o título do item
8      def exibir(self):
9          print(f"Item: {self.titulo}")
10
11 # Definindo a classe derivada Livro que herda de ItemDeBiblioteca
12 class Livro(ItemDeBiblioteca):
13     # Construtor específico para a classe Livro
14     def __init__(self, titulo, autor):
15         super().__init__(titulo) # Chamando o construtor da classe base
16         self.autor = autor # Atributo específico da classe Livro
17
18     # Método para exibir detalhes do livro
19     def exibir(self):
20         print(f"Livro: {self.titulo} - Autor: {self.autor}")
21
22 # Definindo a classe derivada DVD que herda de ItemDeBiblioteca
23 class DVD(ItemDeBiblioteca):
24     # Construtor específico para a classe DVD
25     def __init__(self, titulo, duracao):
26         super().__init__(titulo) # Chamando o construtor da classe base
27         self.duracao = duracao # Atributo específico da classe DVD
28
29     # Método para exibir detalhes do DVD
30     def exibir(self):
31         print(f"DVD: {self.titulo} - Duração: {self.duracao} min")
32

```

**Figura 18:** Exemplo de Herança

Em nosso exemplo, **Livro** e **DVD** herdam de **ItemDeBiblioteca**. O método **super().\_\_init\_\_(titulo)** é utilizado para chamar o construtor da classe base. Ambas as classes derivadas têm o método **exibir()** que sobrescreve o método da classe base. A Figura 19 ilustra a herança em execução.

```

class.py > ...
33 # Testando a herança
34 meu_livro = Livro("1984", "George Orwell") # Criando um objeto da classe Livro
35 meu_dvd = DVD("Matrix", 136) # Criando um objeto da classe DVD
36
37 meu_livro.exibir() # Chamando o método exibir específico da classe Livro
38 # Saída esperada: Livro: 1984 - Autor: George Orwell
39
40 meu_dvd.exibir() # Chamando o método exibir específico da classe DVD
41 # Saída esperada: DVD: Matrix - Duração: 136 min
42

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

PS C:\xampp\htdocs\python_kivy> & C:/Users/lincx/AppData/Local/Programs/Python/Python311/python.exe
c:/xampp/htdocs/python_kivy/classe.py
Livro: 1984 - Autor: George Orwell
DVD: Matrix - Duração: 136 min
PS C:\xampp\htdocs\python_kivy>

```

**Figura 19:** Saída do exemplo de Herança

Os comentários descritos em nossos exemplos, visam uma explicação mais detalhada e clarificada sobre o funcionamento dos códigos, garantindo que as nuances de programação implementadas possam ser melhor compreendidas.

**Nota:**

Trechos após a inserção do símbolo de cerquilha "#", são interpretados como comentários pelo interpretador da linguagem **Python** e não fazem parte da execução do código. Servindo apenas como guias para os desenvolvedores inserirem comentários em seus programas, para si e/ou para outros desenvolvedores que os utilizarão, e que com isso, terão seu entendimento facilitado.

## Polimorfismo

O polimorfismo é frequentemente referido como o terceiro pilar da POO, ao lado do encapsulamento e da herança. A palavra **polimorfismo** vem do grego "polys", que significa "muitos", e "morphē", que significa "forma". Neste contexto, refere-se à capacidade de uma única função ou método ser usado de diferentes maneiras para diferentes tipos de objetos (Deitel & Deitel, 2019).

O polimorfismo torna os programas mais flexíveis e reutilizáveis, pois permite que as classes derivadas implementem e alterem métodos que já estão definidos nas classes base. Esse conceito está intimamente ligado à ideia de herança, mas enquanto a herança estabelece um relacionamento do tipo "**é um**" entre a classe base e a derivada, o polimorfismo enfoca no como as classes derivadas podem ser tratadas como se fossem da classe base (Sweigart, 2019).

Em **Python**, é importante destacar que o polimorfismo é inerente à linguagem devido à sua tipagem dinâmica. Isso significa que a mesma operação pode ser aplicada a diferentes tipos de objetos, e a determinação de qual método ou função usar é feita em tempo de execução, e não em tempo de compilação como em algumas outras linguagens (Martelli et al., 2005).

Para ilustrar o polimorfismo, retomaremos ao exemplo de uma biblioteca. Onde supondo que temos diferentes classes, como **Livro** e **DVD**, ambas derivadas da classe **ItemDeBiblioteca** e ambas com um método **exibir()**. A ideia é que, independentemente do tipo de item, o método **exibir()** mostrará detalhes específicos para aquele item.

Vamos considerar que a nossa biblioteca contém diversos itens, como livros e DVDs. Cada tipo de item tem uma forma única de exibido, mas todos eles, sendo itens da biblioteca, devem ter a capacidade de ser exibidos. Aqui, usaremos o polimorfismo para alcançar essa funcionalidade.

```

classe.py > ...
1  # Classe base para todos os itens da biblioteca
2  class ItemDeBiblioteca:
3      def exibir(self):
4          pass
5
6  # Classe Livro, derivada de ItemDeBiblioteca
7  class Livro(ItemDeBiblioteca):
8      # Construtor para a classe Livro
9      def __init__(self, titulo, autor):
10         self.titulo = titulo # Atributo título para o Livro
11         self.autor = autor # Atributo autor para o Livro
12
13     # Método polimórfico para exibir detalhes específicos do livro
14     def exibir(self):
15         print(f"Livro: {self.titulo} - Autor: {self.autor}")
16
17 # Classe DVD, derivada de ItemDeBiblioteca
18 class DVD(ItemDeBiblioteca):
19     # Construtor para a classe DVD
20     def __init__(self, titulo, duracao):
21         self.titulo = titulo # Atributo título para o DVD
22         self.duracao = duracao # Atributo duração para o DVD
23
24     # Método polimórfico para exibir detalhes específicos do DVD
25     def exibir(self):
26         print(f"DVD: {self.titulo} - Duração: {self.duracao} min")
27

```

**Figura 20:** Exemplo de Polimorfismo

Nas classes implementadas, tanto **Livro** quanto **DVD** têm seu próprio método **exibir()**. Isso é um exemplo de polimorfismo, onde a função tem várias formas. Na Figura 21 a seguir, podemos visualizar o polimorfismo em ação.

```

class.py > ...
28     # Função para mostrar qualquer item da biblioteca
29     def mostrar_item(item):
30         item.exibir() # Chama o método exibir() do objeto fornecido
31
32     # Criando um livro e um DVD
33     meu_livro = Livro("1984", "George Orwell")
34     meu_dvd = DVD("Matrix", 136)
35
36     # Usando a função mostrar_item() para exibir detalhes
37     mostrar_item(meu_livro) # Saída: Livro: 1984 - Autor: George Orwell
38     mostrar_item(meu_dvd)   # Saída: DVD: Matrix - Duração: 136 min
39
PROBLEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE
PS C:\xampp\htdocs\python_kivy> & C:/Users/lincx/AppData/Local/Programs/Python/Python311/python.exe
c:/xampp\htdocs\python_kivy/classe.py
Livro: 1984 - Autor: George Orwell
DVD: Matrix - Duração: 136 min
PS C:\xampp\htdocs\python_kivy>

```

**Figura 21:** Saída do exemplo de polimorfismo

Os comentários detalham trechos do código explicando o propósito das classes, métodos e funções em nosso exemplo. Com elas, procuramos deixar ainda mais clara a capacidade de usar a mesma função `mostrar_item()`, para exibir tanto livros quanto DVDs e, portanto, demonstrarmos de forma prática a essência do polimorfismo.

## Aplicação da POO em Smartphones com Python e Kivy

### Criar Classes de Interface de Usuário

Você pode criar classes que representam elementos de interface do usuário, como botões, janelas, caixas de texto etc. Essas classes podem herdar características de classes base do *Kivy* para possuírem em si, comportamentos específicos adicionados.

```

from kivy.uix.button import Button

class MeuBotao(Button):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.text = "Clique-me!"

```

**Figura 22:** Exemplo de herança dos comportamentos de botões presentes (já implementados) no pacote `kivy.uix.button`

### Definir Comportamento de Eventos

Você pode definir métodos nas classes de interface de usuário para manipular eventos, como cliques de botão (direito, esquerdo, scroll, etc.). O que permitirá que você associe comportamentos específicos a objetos de interface do usuário.

```

class MeuBotao(Button):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.text = "Clique-me!"
        self.bind(on_press=self.on_button_click)

    def on_button_click(self, instance):
        self.text = "Clicou!"

```

**Figura 23:** Exemplo de comportamento de Eventos

### Criar Objetos de Interface de Usuário

Você cria objetos de interface de usuário instanciando suas classes. Por exemplo:

```
botao1 = MeuBotao()
botao2 = MeuBotao()
```

Figura 24: Exemplo de criação de Objetos

### Gerenciar a Interface de Usuário

Use um *layout* da classe **uix** do Kivy, como o **BoxLayout** ou o **GridLayout** para organizar os objetos de interface do usuário em tela. Adicione então esses objetos ao *layout* para exibi-los.

```
from kivy.uix.boxlayout import BoxLayout

layout = BoxLayout()
layout.add_widget(botao1)
layout.add_widget(botao2)
```

Figura 25: Exemplo de uso de layout com **BoxLayout**

### Herança e Reutilização de Código

Você pode criar classes personalizadas que herdam de classes Kivy existentes para criar *widgets* personalizados com base em *widgets* padrão.

```
from kivy.uix.label import Label

class MeuLabel(Label):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.text = "Isso é um rótulo personalizado."
```

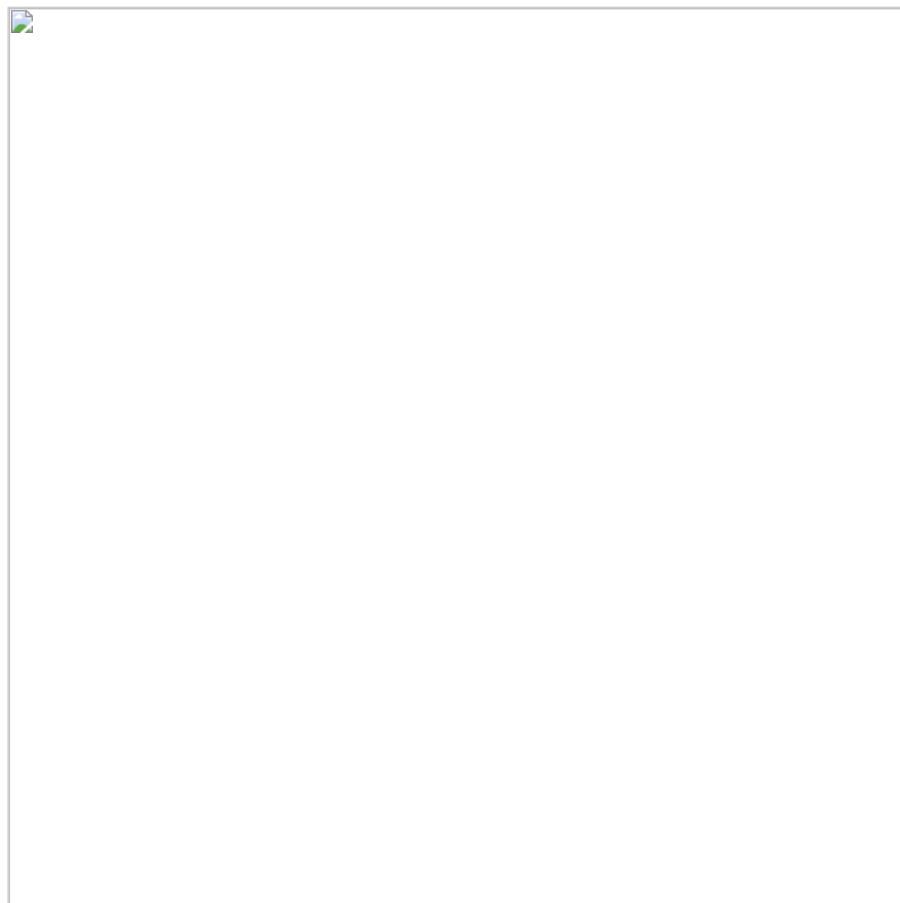
Figura 26: Herança e Reutilização de Código

Em nosso exemplo, os dois asteriscos **\*\*** antes de **kwargs** indicam que a função aceita qualquer número de argumentos de palavra-chave em forma de dicionário, portanto **kwargs** é um dicionário que contém todos os argumentos nomeados que foram passados.

Já a linha **self.text = "Isso é um rótulo personalizado."**: define o atributo **text** da instância do **MeuLabel** para uma *string* literal diz “Isso é um rótulo personalizado.”. Em Kivy, o atributo **text** de um **Label** determina o texto que será exibido pelo *widget* na interface do usuário.

### Polimorfismo

Aproveite o polimorfismo para criar funções que aceitem objetos de diferentes classes de *widgets* Kivy como argumentos.



**Figura 27:** Exemplo de polimorfismo

No desenvolvimento de aplicativos móveis com *Python* e *Kivy*, a POO é fundamental para criar uma estrutura organizada e manutenível para a interface de usuário e o comportamento do aplicativo. Ao implementar classes personalizadas usando herança e polimorfismo, você pode criar aplicativos móveis flexíveis e escaláveis.

#### Exemplo de Aplicação

A seguir trazemos um exemplo de aplicação móvel usando *Python* e o framework *Kivy*, o qual demonstra a POO aplicada à *smartphones*. Nele criamos um aplicativo que exibe um botão na tela e, quando o botão for clicado, será exibida uma mensagem.

Para executar nosso exemplo de forma correta, antes de começarmos certifique-se de que você tem o *Kivy* instalado no seu ambiente de desenvolvimento. Para tal, basta executar os passos a seguir:

- Primeiro, verifique via terminal qual é a versão *Python* instalada em sua máquina.
  - Comando: `python -- version`
- Segundo (se ainda não o fez), instale o *Kivy* - neste caso, utilizamos o *pip* via terminal para realizar tal ação.
  - Comando: `pip3 install kivy`

A Figura 28 a seguir, ilustra as saídas de execução das linhas inseridas no terminal do IDE *VsCode*.

```

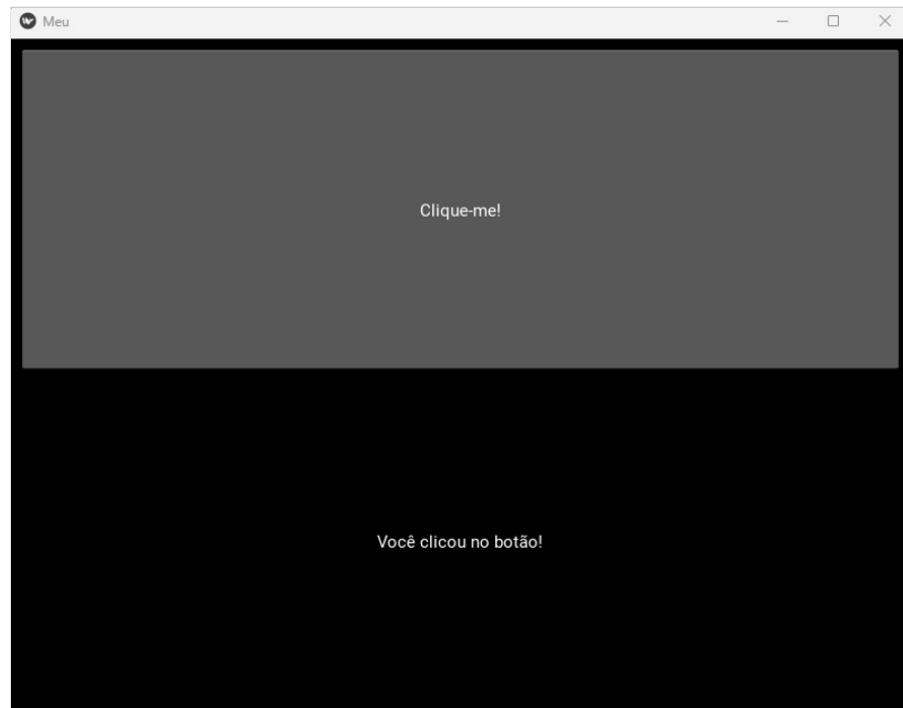
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE + ... ^ X
PS C:\xampp\htdocs\python_kivy> & C:/Users/lincx/AppData/Local/Programs/Python/Python311/python.exe c:/xampp/htdocs/python_kivy/teste.py
PS C:\xampp\htdocs\python_kivy> python --version
Python 3.11.5
PS C:\xampp\htdocs\python_kivy> pip3 install kivy
Ln 1, Col 1  Spaces: 4  UTF-8  CRLF  Python  3.11.5 64-bit
  
```

**Figura 28:** Demonstração dos comandos inseridos no terminal do *VSCode*

```

pyton_kivy > python_kivy.py > ...
1  from kivy.app import App
2  from kivy.uix.boxlayout import BoxLayout
3  from kivy.uix.button import Button
4  from kivy.uix.label import Label
5
6  class MeuApp(App):
7      def build(self):
8          # Cria um layout BoxLayout vertical para organizar os elementos na tela.
9          layout = BoxLayout(orientation='vertical', padding=10, spacing=10)
10
11         # Cria um botão e o vincula ao método mostrar_mensagem quando pressionado.
12         self.botao = Button(text="Clique-me!")
13         self.botao.bind(on_press=self.mostrar_mensagem)
14
15         # Adiciona o botão ao layout.
16         layout.add_widget(self.botao)
17
18         # Cria um rótulo para exibir mensagens.
19         self.label = Label(text="Pressione o botão")
20
21         # Adiciona o rótulo ao layout.
22         layout.add_widget(self.label)
23
24         # Retorna o layout como a interface principal do aplicativo.
25         return layout
26
27     def mostrar_mensagem(self, instance):
28         # Atualiza o texto do rótulo quando o botão é clicado.
29         self.label.text = "Você clicou no botão!"
30
31 if __name__ == "__main__":
32     # Inicia o aplicativo.
33     MeuApp().run()
34

```

**Figura 29:** Exemplo de aplicação com Kivy**Figura 30:** Interface gráfica de layout do tipo **BoxLayout** gerada ao executar o código da aplicação

#### Explicação do código:

- Um *layout BoxLayout* vertical é criado para organizar elementos na tela.
- Um botão **self.botao** é criado e vinculado ao método **self.mostrar\_mensagem** para manipular o evento de pressionar o botão.
- O botão é adicionado ao *layout*.
- Um rótulo **self.label** é criado para exibir mensagens.

- O rótulo também é adicionado ao *layout*.
- O *layout* é retornado como a interface principal do aplicativo.
- O método **self.mostrar\_mensagem** atualiza o texto do rótulo quando o botão é clicado.
- O aplicativo é iniciado com **MeuApp().run()**

Agora, os elementos de interface do usuário são organizados de forma adequada no *layout* vertical, e a mensagem é exibida no quando o botão é pressionado. Isso cria uma experiência de interface de usuário mais clara e organizada no aplicativo.

## Desenvolvimento de Aplicativos para Smartphones - Parte I

Desenvolver aplicativos para *smartphones* é uma atividade empolgante e desafiadora que envolve várias considerações e técnicas. Nesta seção iremos explorar os principais aspectos do desenvolvimento de aplicativos móveis.

### a. Interface do Usuário (UI - User Interface) e Experiência do Usuário (UX - User eXperience)

A interface do usuário (UI) e a experiência do usuário (UX) são elementos essenciais no *design* de produtos digitais. UI engloba a visual e a interação, enquanto UX se refere à forma como o usuário percebe e interage com o produto. Ambas são fundamentais para proporcionar uma experiência satisfatória e impactar positivamente nos resultados do negócio.

A interface do usuário (UI) e a experiência do usuário (UX) são considerados aspectos críticos no desenvolvimento de aplicativos móveis. Uma UI bem projetada e uma UX agradável são essenciais para atrair e manter os usuários. A seguir, citamos alguns pontos importantes para ambas:

#### Design Responsivo

Certifique-se de que sua interface se adapte a diferentes tamanhos de tela e orientações (vertical e horizontal).

#### Navegação Intuitiva

Projete um fluxo de navegação lógico e fácil de entender para os usuários.

#### Design Visual Atraente

Use cores, ícones e elementos de *design* que sejam atraentes e estejam em conformidade com a identidade visual do aplicativo.

#### Feedback do Usuário

Forneça *feedback* claro quando os usuários interagirem com elementos da interface, como botões clicados, ícones, mensagens de retorno, etc.

#### Usabilidade

Garanta que os elementos da interface sejam acessíveis e fáceis de usar, mesmo para usuários com diferentes níveis de experiência.

A Figura 31 a seguir, descreve algumas tendências comuns em UI e UX.

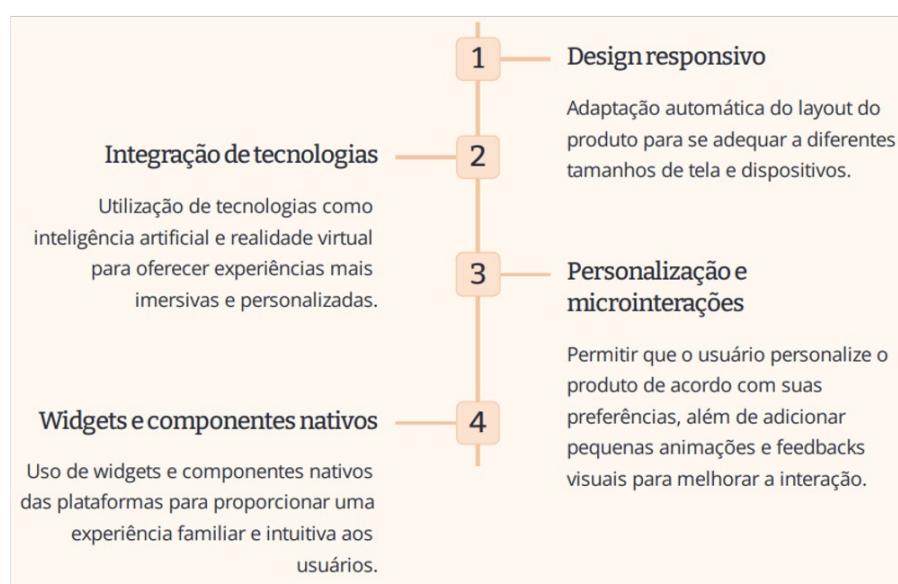


Figura 31: Tendências em UI e UX (FDCEa, 2023)

### b. Widgets e Componentes Nativos

Os aplicativos móveis são compostos por uma variedade de *widgets* e componentes nativos que permitem aos usuários interagir com o aplicativo. Esses componentes podem variar dependendo do SO (*Android* ou *iOS*). Alguns exemplos comuns incluem:

- **Botões**: Para ações do usuário, como enviar formulários ou navegar entre telas.
- **Caixas de Texto**: Para entrada de texto, como digitar mensagens ou pesquisar.
- **Listas e Grades**: Para exibir listas de itens ou informações.
- **Menus**: Para acesso a funcionalidades adicionais ou opções de configuração.
- **Mapas**: Para exibir informações geográficas e direções.

### c. Layouts e Recursos de Design

Os *layouts* definem como os elementos da interface do usuário são organizados na tela. Alguns tipos comuns de *layouts* incluem

- **LinearLayout**: Organiza elementos em uma única coluna ou linha.
- **RelativeLayout**: Define a posição dos elementos em relação a outros elementos.
- **GridLayout**: Organiza elementos em uma grade de células.
- **ConstraintLayout**: Permite criar *layouts* complexos com base em restrições.

A Figura 32 a seguir, detalha alguns itens constantes no processo de *design*.

<b>Ideação</b>	Compreender as necessidades do usuário e gerar ideias para o design do produto.
<b>Prototipação</b>	Criar protótipos que permitam visualizar a aparência e a funcionalidade do produto.
<b>Testes</b>	Realizar testes com usuários para validar o design e identificar possíveis melhorias.
<b>Iteração</b>	Realizar alterações com base nos feedbacks dos usuários e repetir o ciclo de prototipação e teste.

Figura 32: Itens do processo de *design* (FDCEb, 2023)

Além disso, os recursos de *design*, como cores, tipografia e ícones, desempenham um papel crucial na criação de uma identidade visual consistente para os aplicativos.

Para o desenvolvimento da interface de nossos aplicativos, entre outros, podemos optar entre os pacotes *Kivy* e *KivyMD* <<https://pypi.org/project/kivymd/#files>>, mas isso irá depender das necessidades específicas do projeto e das preferências pessoais de quem o estiver implementando. Sendo que nos caso dos *frameworks* citados, ambos são poderosos e têm vantagens específicas.

A seguir citamos as principais características de cada um deles:

#### *Kivy*

É um *framework* de desenvolvimento de aplicativos multiplataforma altamente flexível e personalizável. Ele não é limitado a *designs* específicos e oferece controle total sobre a aparência e o comportamento dos elementos da interface.

- **Interface Personalizada**: Com o *Kivy* você tem a liberdade de criar interfaces de usuário totalmente personalizadas, o que é ideal se você deseja um *design* único e não deseja se limitar a um conjunto específico de componentes de interface.

- **Grande Comunidade:** O Kivy possui uma comunidade ativa e numerosa, além de uma vasta quantidade de recursos e documentação disponíveis. Isso pode ser útil para solucionar problemas e aprender mais e facilmente sobre o framework.

### KivyMD (KivyMaterial Design)

**Baseado no Material Design:** O KivyMD é um conjunto de extensões para o Kivy que implementa os princípios do *Material Design*, que o torna ideal para criar aplicativos com uma aparência moderna e consistente de acordo com as diretrizes do Google.

**Componentes Prontos para Uso:** O KivyMD fornece uma ampla variedade de componentes de interface de usuário, como botões, barras de navegação, cartões, caixas de diálogo, etc., que são projetados com base no *Material Design*. Isso economiza tempo e esforço ao criar interfaces modernas e minimalistas.

**Facilidade de Uso:** Para desenvolvedores que desejam uma experiência de desenvolvimento mais rápida e direta, o KivyMD oferece componentes prontos para uso que seguem os padrões de design do *Material Design* <<https://m3.material.io/>>.

A escolha entre Kivy e KivyMD depende de suas preferências de design, do estilo visual desejado para seu aplicativo e da flexibilidade que você precisa. Portanto se você está criando um aplicativo que deve seguir rigorosamente as diretrizes do *Material Design* e deseja uma implementação rápida e fácil de componentes de interface, o KivyMD pode ser a escolha ideal.

No entanto, se você precisa de um alto nível de personalização e flexibilidade na criação de interfaces de usuário, o Kivy "puro" pode ser mais adequado. Também é possível combinar os dois, usando Kivy para partes altamente personalizadas e KivyMD para componentes *Material Design* específicos, dependendo das necessidades do projeto.

### Para Aprender e Treinar

Como já esclarecido, em nosso curso optamos por utilizar o framework Kivy para desenvolvermos nossos exemplos, todavia podemos ser bastante útil e instigante conhecer também as possibilidades oferecidas pelo KivyMD. Para tal, sugerimos que você assista atentamente ao vídeo intitulado "**KivyMD, aparência nativa para AppsAndroid**", o qual está disponível no YouTube e mostra em detalhes como o framework deve ser instalado e posteriormente ter vários de seus recursos explorados.

<sup>1</sup> Um sistema de design - apoiado em código-fonte aberto, que ajuda equipes a criarem experiências digitais de alta qualidade. Representa um conjunto de diretrizes (*guidelines*) criadas pela Google LLC. para padronizar todas as suas interfaces gráficas.

### d. Manipulação de Eventos (Toques, Gestos, etc.)

Os aplicativos móveis respondem a eventos gerados pelos usuários. Isso inclui toques na tela, gestos como arrastar e soltar, inclinação do dispositivo e muito mais. A manipulação de eventos é implementada por meio de código para que o aplicativo responda corretamente às interações do usuário.

Por exemplo, um evento de toque pode ser usado para abrir um menu, um gesto de deslizar para navegar entre páginas e um sentido de movimento para jogos baseados em movimento.

### e. Exemplo de Aplicação

Aqui está um exemplo simples de aplicativo móvel em Python com o framework Kivy que demonstra alguns dos princípios citados. Figura 33 a seguir, ilustra o código implementado de um aplicativo que exibe uma lista de itens em uma interface do usuário básica.

```
python.kivy > app.py
 1  from kivy.app import App
 2  from kivy.uix.boxlayout import BoxLayout
 3  from kivy.uix.button import Button
 4  from kivy.uix.textinput import TextInput
 5  from kivy.uix.label import Label
 6
 7  class ListaDeTarefas(App):
 8      def build(self):
 9          self.tarefas = [] # Lista para armazenar as tarefas
10          layout = BoxLayout(orientation='vertical', padding=10, spacing=10)
11
12          self.input_tarefa = TextInput(hint_text="Digite uma tarefa")
13          layout.add_widget(self.input_tarefa)
14
15          adicionar_botaao = Button(text="Adicionar Tarefa")
16          adicionar_botaao.bind(on_press=self.adicionar_tarefa)
17          layout.add_widget(adicionar_botaao)
18
19          self.lista_layout = BoxLayout(orientation='vertical', spacing=5)
20          layout.add_widget(self.lista_layout)
21
22      return layout
23
24  def adicionar_tarefa(self, instance):
25      texto_tarefa = self.input_tarefa.text
26      if texto_tarefa:
27          tarefa_label = Label(text=texto_tarefa)
28          self.lista_layout.add_widget(tarefa_label)
29          self.tarefas.append(texto_tarefa)
30          self.input_tarefa.text = ""
31
32  if __name__ == "__main__":
33      ListaDeTarefas().run()
34
```

Figura 33: Exemplo de aplicação em Python com Kivy

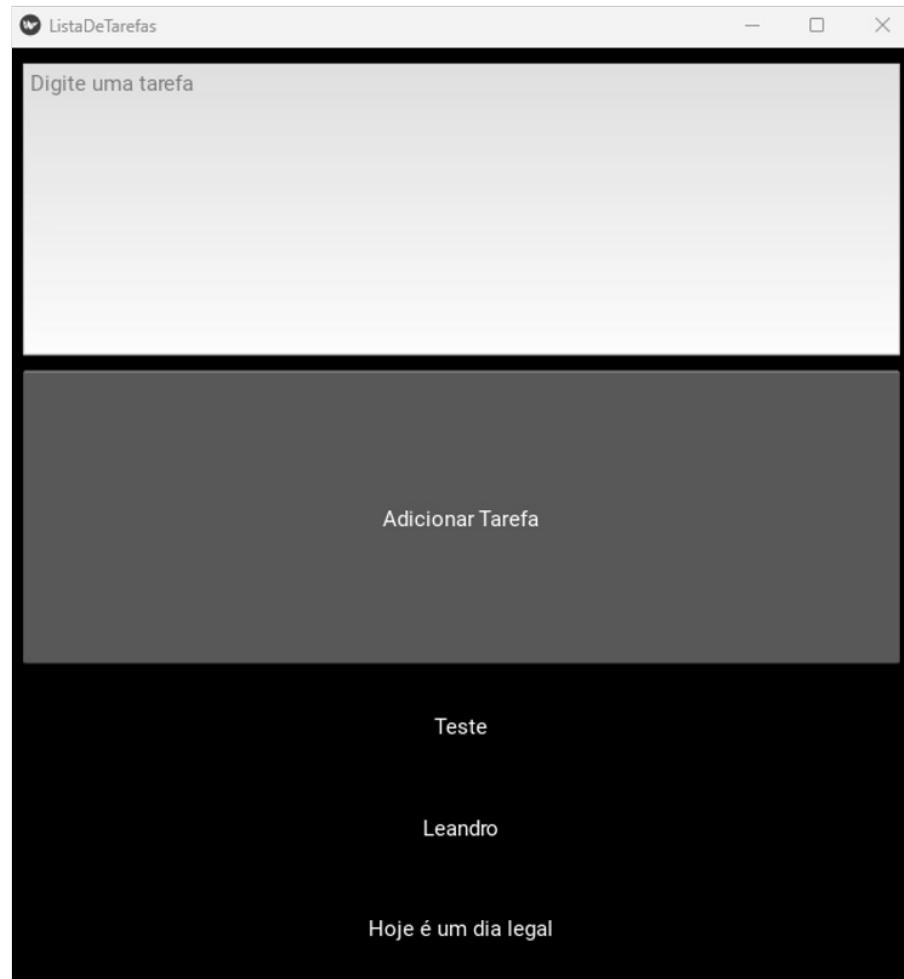


Figura 34: Tela do aplicativo gerado

## Desenvolvimento de Aplicativos para Smartphones - Parte II

Continuando nossa exploração do desenvolvimento de aplicativos para *smartphones*, vamos abordar nesta seção, tópicos importantes relacionados à persistência de dados, armazenamento, comunicação entre aplicativos, integração com serviços *Web*, apresentar exemplos de aplicação e discutir várias questões relevantes.

### a. Persistência de Dados

A persistência de dados é essencial para armazenar informações importantes em aplicativos móveis. Isso permite que os dados sobrevivam além do ciclo de vida do aplicativo. A seguir apresentamos um exemplo de como usar o *SQLite* <<https://www.sqlite.org/draft/download.html>> para persistência de dados em *Python* com *Kivy*.

```
python_kivy > sqlite.py
1 import sqlite3
2
3 # Conectar ao banco de dados SQLite
4 conn = sqlite3.connect('meu_banco_de_dados.db')
5
6 # Criar uma tabela para armazenar tarefas
7 conn.execute('''CREATE TABLE IF NOT EXISTS tarefas
8 | | | (id INTEGER PRIMARY KEY AUTOINCREMENT,
9 | | | descricao TEXT);''')
10
11 # Inserir uma tarefa no banco de dados
12 conn.execute("INSERT INTO tarefas (descricao) VALUES ('Comprar leite');")
13
14 # Consultar as tarefas
15 cursor = conn.execute("SELECT id, descricao FROM tarefas;")
16 for row in cursor:
17     print(f"ID: {row[0]}, Tarefa: {row[1]}")
18
19 # Fechar a conexão com o banco de dados
20 conn.close()
21
```

Figura 35: *SQLite* para persistência de dados em *Python*

### b. Armazenamento de Dados Locais e Remotos

**Local:** Usar **SharedPreferences** (*Android*) ou **UserDefaults** (*iOS*) para armazenar configurações ou preferências do usuário localmente.

```
# Exemplo de armazenamento local com Kivy (Python)
from kivy.storage.jsonstore import JsonStore

# Criar um armazenamento JSON local
store = JsonStore('configuracoes.json')

# Salvar uma configuração
store.put('config1', valor='valor_da_configuracao')

# Recuperar uma configuração
valor = store.get('config1')['valor']
```

Figura 36: Exemplo de armazenamento local com *Kivy* (*Python*)

**Remoto:** Utilize bibliotecas como **requests** ou **http.client** para fazer solicitações a APIs *Web* e processar as respostas.

```

import requests

# Exemplo de solicitação HTTP GET com Python
response = requests.get('https://api.example.com/dados')

# Processar a resposta JSON
if response.status_code == 200:
    dados = response.json()
    print(dados)

```

Figura 37: Exemplo de solicitação HTTP GET com Python

### c. Comunicação entre Aplicativos

**Intenções (Android):** Abra outro aplicativo ou envie informações entre aplicativos Android.

```

from jnius import autoclass

# Criar uma intenção
Intent = autoclass('android.content.Intent')
intent = Intent()
intent.setAction(Intent.ACTION_SEND)
intent.putExtra(Intent.EXTRA_TEXT, "Texto para compartilhar")
intent.setType("text/plain")

# Iniciar a atividade
currentActivity.startActivity(intent)

```

Figura 38: Comunicação entre aplicativos Android

**URL Schemes (iOS):** Abra outro aplicativo ou comunique-se com outros aplicativos iOS.

```

import webbrowser

# Abrir um aplicativo com URL Scheme
url_scheme = "appname://"
webbrowser.open(url_scheme)

```

Figura 39: Comunicação entre aplicativos iOS

### Saiba Mais

Antes de prosseguirmos, sugerimos que você assista a dois vídeos disponíveis no *YouTube*, os quais trarão além da demonstração de exemplos práticos em execução, novas perspectivas de utilização de diferentes recursos (ferramentas, SOs e dispositivos) em seus próximos aplicativos.

[Transformando Aplicativos Python em APK Android](#)

e

[Python Kivy - Como gerar um APK!](#)

#### d. Integração com Serviços Web

##### Notificação por Push: Uma Visão Geral

A notificação por *push* é uma técnica de comunicação que permite que aplicações móveis, *desktops* ou *sites* enviem informações diretamente para o dispositivo do usuário sem a necessidade de que mesmo solicite tais informações ativamente. Em outras palavras, essas notificações "empurram" conteúdo ou alertas para o dispositivo do usuário, sem que ele precise abrir a aplicação para recebê-las. Esta subseção irá explorar o funcionamento, a finalidade e as vantagens das notificações *push*.

##### Como isso funciona?

- **Registro/Inscrição:** Para começar, o dispositivo do usuário deve estar inscrito para receber notificações *push* de uma aplicação específica. Geralmente, isso ocorre quando o usuário instala um aplicativo e concorda em receber notificações ou quando visita um site que oferece notificações *push* e aceita recebê-las.
- **Identificação do Dispositivo:** Uma vez inscrito, o serviço de notificação *push* gera um *token* único para o dispositivo, que funciona como um endereço para onde as notificações serão enviadas.
- **Envio da Notificação:** Quando a aplicação precisa enviar uma notificação, seja ela um aplicativo móvel, um *site* ou um sistema, faz uma requisição ao serviço de notificação *push* (como *Firebase Cloud Messaging* para Android ou *Apple Push Notification Service* para dispositivos iOS) com a mensagem e o *token* do dispositivo.
- **Entrega ao Dispositivo:** O serviço de notificação *push* encaminha a mensagem para o dispositivo correspondente com base no *token* fornecido. Se o dispositivo estiver *offline* no momento do envio, a mensagem será armazenada e entregue assim que o dispositivo estiver *online* novamente.

##### Para que serve?

- **Engajamento do Usuário:** As notificações *push* são ferramentas poderosas para reengajar usuários, lembrando-os de usar um aplicativo ou visitar um *site*.
- **Comunicação Direta:** As empresas podem se comunicar diretamente com os usuários, fornecendo atualizações, ofertas especiais ou notícias importantes.
- **Entrega de Conteúdo em Tempo Real:** Esportes ao vivo, resultados de eleições ou alertas de emergência podem ser entregues em tempo real através de notificações *push*.
- **Personalização:** As notificações podem ser personalizadas com base no comportamento, localização ou preferências do usuário, tornando-as mais relevantes.

##### Vantagens

- **Imediatismo:** As notificações são entregues instantaneamente, permitindo comunicação em tempo real.
- **Alta Visibilidade:** Ao contrário dos *e-mails* que podem acabar em pastas de *spam*, as notificações *push* têm alta visibilidade, aparecendo diretamente na tela do dispositivo.
- **Taxas de Clique:** Estudos indicam que notificações *push* tendem a ter taxas de cliques mais altas do que outras formas de comunicação digital, como *e-mails*.

Embora as notificações *push* ofereçam uma série de benefícios, é crucial usá-las com moderação e consideração. O envio excessivo ou de notificações irrelevantes pode levar os usuários a se desinscreverem ou desinstalarem um aplicativo. Por isso, é sempre recomendado entender bem seu público-alvo e oferecer valor real através de cada notificação enviada.

A integração com serviços *Web* é comum para acessar recursos como autenticação de usuário, notificações *push*, armazenamento na nuvem e muito mais. Você pode usar bibliotecas de cliente *HTTP* para fazer solicitações a APIs *Web* e processar as respostas, conforme mostrado anteriormente.

#### e. Exemplo de Aplicação

Vamos considerar um exemplo de aplicativo de lista de tarefas que incorpora os conceitos trazidos até aqui. A Figura 40 a seguir ilustra um trecho de código necessário para tal.

```

# Abrir um aplicativo com URL Scheme
url_scheme = "appname://"
webbrowser.open(url_scheme)

# Exemplo de aplicativo de lista de tarefas com Kivy (Python)
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput

class ListaDeTarefas(App):
    def build(self):
        # Código de construção da interface de usuário
        pass

    def adicionar_tarefa(self, instance):
        # Lógica para adicionar tarefas
        pass

if __name__ == "__main__":
    ListaDeTarefas().run()

```

**Figura 40:** Trecho de código com integração de serviços Web

Durante o desenvolvimento de aplicativos para *smartphones*, você pode enfrentar diversas questões:

- **Segurança:** Certifique-se de proteger os dados do usuário e garantir uma comunicação segura com serviços Web.
- **Compatibilidade de Plataforma:** Considere as diferenças entre Android e iOS, como *design* de interface e APIs específicas.
- **Otimização de Desempenho:** Garanta que o aplicativo funcione de maneira eficiente em todos os dispositivos.
- **Manutenção:** Continue atualizando e aprimorando o aplicativo à medida que as plataformas evoluem.
- **Testes:** Realize testes rigorosos em diferentes dispositivos e sistemas operacionais.

Um passo importante no desenvolvimento de aplicativos para *smartphones* é a publicação do aplicativo nas respectivas lojas de aplicativos. Vamos discutir a seguir como ocorre o processo de publicação, desde a preparação até a disponibilização do aplicativo para o usuário.

## Processo de Publicação de Aplicativos

### a. Preparação para o lançamento

Antes de publicar um aplicativo, você deve se preparar para atender aos requisitos das lojas de aplicativos e assim fornecer a melhor experiência de usuário possível:

- **Teste de Qualidade:** Execute testes rigorosos em diferentes dispositivos e SOs para garantir que seu aplicativo funcione sem problemas (Fitzgerald, 2021).
- **Otimização de Performance:** Assegure-se de que seu aplicativo seja ágil e eficiente, reduzindo tempos de carregamento e otimizando imagens e outros recursos.
- **Política de Privacidade:** Em muitas jurisdições, ter uma política de privacidade é obrigatório, especialmente se você coletar informações do usuário (Rosenberg & Mateescu, 2022).
- **Gráficos e Mídias:** Prepare imagens de alta qualidade para ícones, capturas de tela e vídeos promocionais. Estes são itens criados para atrair usuários nas lojas de aplicativos.
- **Classificação Etária:** De acordo com o conteúdo e a funcionalidade do seu aplicativo, escolha a classificação etária adequada.

### b. Configurando os estabelecimentos de aplicativos

Ambas as principais lojas de aplicativos, *Google Play* e *Apple App Store*, têm suas diretrizes específicas. Cada plataforma (Android e iOS) tem sua própria loja de aplicativos e para publicá-lo(s), você precisará criar uma listagem do(s) mesmo(s) e configurar uma conta.

desenvolvedor:

**Conta de Desenvolvedor:** Crie uma conta de desenvolvedor na *Google Play Store* (Android) ou na *App Store* da *Apple* (iOS). Isso geralmente envolve uma taxa única ou uma taxa anual.

**Google Play:** Você precisará de uma conta de desenvolvedor *Google Play*, que tem uma taxa única. Certifique-se de seguir o Guia de Lançamento de Aplicativos fornecido pela *Google LLC*. (Google, 2020).

**Apple App Store:** A inscrição no Programa de Desenvolvedores da *Apple* tem uma taxa anual. Os aplicativos são submetidos a revisão mais rigorosa na *App Store*, então esteja preparado(a) para seguir tais diretrizes com atenção (Apple, 2021).

**Detalhes do Aplicativo:** Forneça informações detalhadas sobre o aplicativo, incluindo nome, descrição, imagens, categorias e classificação etária.

**Preço e Distribuição:** Determine se o aplicativo será gratuito ou pago e configure os países e regiões onde deseja disponibilizá-lo.

**Capturas de Tela e Recursos Gráficos:** Faça o *upload* das capturas de tela e recursos gráficos necessários, como ícones e imagem de tela inicial.

**Política de Privacidade:** Forneça um *link* para a política de privacidade do aplicativo, se aplicável.

### c. Processo de Publicação

Após configurar sua conta de desenvolvedor e listar todas as informações necessárias, você pode iniciar o processo de publicação.

**Envio do Aplicativo:** Faça o *upload* do arquivo **APK**<sup>1</sup> (Android) ou do pacote de aplicativo (iOS) para a loja de aplicativos correspondente.

**Revisão e Aprovação:** As lojas de aplicativos realizarão uma revisão do seu aplicativo para garantir que ele esteja em conformidade com as suas diretrizes e políticas. Isso pode levar alguns dias.

**Lançamento:** Após a aprovação, você pode agendar o lançamento do aplicativo ou publicá-lo imediatamente.

**Atualizações:** À medida que você faz atualizações no(s) aplicativo(s), repita o processo de envio e revisão para disponibilizar as novas versões.

**Marketing e Promoção:** Promova seu aplicativo por meio de estratégias de marketing, redes sociais e outras ferramentas para aumentar o número de *downloads*.

Quando utilizamos *Python* e *Kivy* para desenvolvimento *mobile*, o processo de publicação se diferencia um pouco das aplicações nativas tradicionais. Vamos detalhar a seguir, tal processo:

#### Empacotamento do Aplicativo

O primeiro passo é transformar seu aplicativo *Kivy* em um APK para Android ou em um IPA<sup>2</sup> para iOS.

##### Para Android:

- Instale o *Buildozer* <<https://pypi.org/project/buildozer/>> ou o *Python for Android*. <<https://pypi.org/project/python-for-android/>>.
- Utilize o *Buildozer* para gerar um arquivo **.spec**, que é uma especificação para construir o aplicativo.
- Modifique o arquivo **.spec** conforme necessário, especificando detalhes como permissões, arquiteturas de destino, e outros.
- Execute o *Buildozer* para compilar e criar o APK.

##### Nota:

Se estiver em um ambiente *MacOS*, pode ser necessário usar uma máquina virtual ou um serviço de construção na nuvem para compilar para Android (Somerville, 2021).

#### Para iOS (no macOS):

- Instale o pacote **toolchain** do *Kivy* para iOS.
- Use a ferramenta para criar um novo projeto e inclua seu aplicativo *Kivy*.
- Use o *Xcode* <<https://developer.apple.com/xcode/resources/>> para finalizar a configuração e compilar o IPA.
- Submeta o aplicativo.

#### Para Google Play:

- Crie ou acesse sua Conta de Desenvolvedor no *Google Play Console*.

- Clique em "Criar aplicativo" e siga as etapas, fornecendo detalhes e metadados para o seu aplicativo.
- Carregue o APK que você gerou com o *Builder*.
- Configure todos os detalhes adicionais, como listagens de lojas, classificações de conteúdo, preços e territórios.

### **Para Apple App Store:**

- Acesse o *App Store Connect*.
- Crie um novo aplicativo e forneça todos os detalhes e metadados necessários.
- Carregue seu IPA usando o *Xcode* ou *Application Loader* <<https://developer.apple.com/xcode/>> para versões anteriores ao *Xcode*.
- Revisão e Aprovação.

Após o envio, seu aplicativo será submetido a uma revisão. Para o *Google Play*, esse processo leva geralmente de algumas horas até dois dias. Já *Apple App Store*, é conhecida por ter um processo de revisão mais rigoroso, e que pode levar de um dia a até uma semana.

### **Lançamento**

#### **Uma vez aprovado:**

- No *Google Play*, você pode optar por um lançamento em fases ou um lançamento completo.
- Na *Apple App Store*, você pode definir a data de lançamento ou torná-lo disponível imediatamente.

Depois que o aplicativo estiver disponível nas lojas de aplicativos, os usuários poderão encontrá-lo, fazer o *download* e começar a usá-lo. Lembre-se de que o processo de publicação pode variar em detalhes específicos entre as lojas, portanto, é importante seguir as diretrizes e recomendações fornecidas por cada plataforma. Além disso, esteja preparado para fornecer suporte e atualizações regulares para garantir uma experiência contínua aos seus usuários.

<sup>1</sup> Sigla para *Android Application Pack*, ou pacote de aplicações que pode ser descompactado e instalado no *Android*. Ele segue o mesmo propósito de arquivos com extensão ".exe", ou seja, arquivos executáveis por exemplo como ocorre nas instalações de programas no *MS-Windows*.

<sup>2</sup> *iOS Package App Store*, pacotes de aplicativos que podem ser instalados em dispositivos da *Apple Inc.*

## **Tópicos Avançados em Programação para Smartphones**

Nesta subseção, vamos explorar alguns tópicos avançados em programação para *smartphones*, incluindo acesso a recursos de *hardware*, programação assíncrona, integração com APIs de terceiros, desenvolvimento de jogos e apresentar exemplos de aplicativos sobre tais tópicos.

À medida que avançamos no mundo do desenvolvimento de aplicativos móveis com *Python* e *Kivy*, é fundamental expandir nossos horizontes e aprofundar em tópicos mais avançados. Estes tópicos, embora possam parecer complexos à primeira vista, podem levar a aplicativos mais eficientes, interativos e envolventes.

### **Programação Assíncrona**

A programação assíncrona permite que os aplicativos executem várias tarefas simultaneamente, sem bloquear a principal *thread* (processo ou tarefa que pode ser executada concorrentemente) do aplicativo. Isso é particularmente útil em aplicativos móveis onde operações demoradas, como acessos a redes ou leitura de grandes volumes de dados, podem prejudicar a experiência do usuário.

A Figura 41 a seguir, ilustra um exemplo prático de uma aplicação que precisa buscar dados de um servidor. Em vez de bloquear a interface enquanto aguarda a resposta, podemos fazer essa solicitação assincronamente.



```

testes.py > ...
1   from kivy.network.urlrequest import URLRequest
2
3   def on_success(req, result):
4       print("Dados recebidos:", result)
5
6   def fetch_data():
7       req = URLRequest('https://api.example.com/data', on_success=on_success)
8

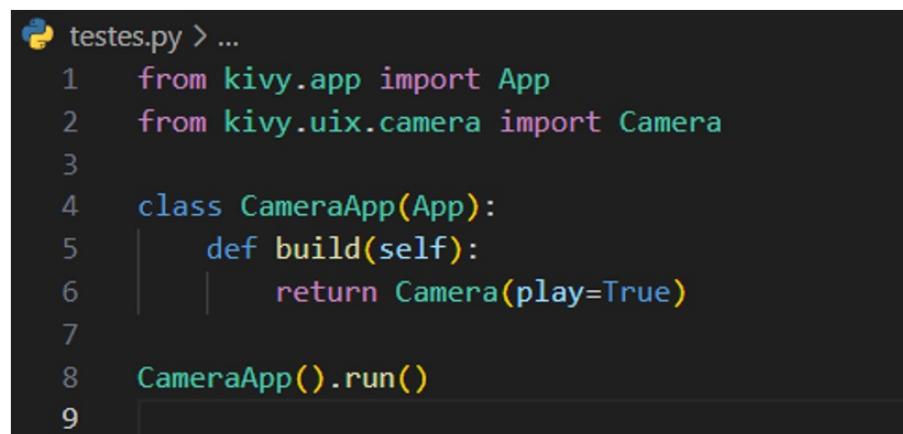
```

Figura 41: Programação assíncrona

Note que **URLRequest** é parte do pacote **network** do *Kivy* e fornece um método assíncrono para fazer solicitações HTTP/1.x.

#### a. Acesso a Recursos de Hardware (Câmera, Sensores, etc.)

*Smartphones* modernos vêm com uma variedade de sensores e componentes de *hardware*, sendo o acesso a tais aos recursos fundamental para criarmos aplicativos avançados. A Figura 42 a seguir, ilustra um exemplo prático de como acessar a câmera (inicia câmera e exibir a sua visualização em tempo real) em *Python* usando *Kivy*.



```

testes.py > ...
1   from kivy.app import App
2   from kivy.uix.camera import Camera
3
4   class CameraApp(App):
5       def build(self):
6           return Camera(play=True)
7
8   CameraApp().run()
9

```

Figura 42: Exemplo de código *Python* para acessar a câmera de um dispositivo

### Integração com APIs de Terceiros

Em um mundo cada vez mais conectado, a integração entre diferentes plataformas e serviços se tornou uma prática comum e indispensável no desenvolvimento de aplicativos. Uma das maneiras mais eficientes de estabelecer essa comunicação é através de

Uma das grandes vantagens da integração com APIs de terceiros é a economia de tempo, pois ao invés de desenvolver uma funcionalidade do zero, os desenvolvedores podem aproveitar recursos de serviços já estabelecidos e amplamente testados. Por exemplo, se um aplicativo precisa de um sistema de pagamento, ele pode integrar-se com APIs de plataformas como *Google Pay*, *Apple Pay*, *PayPal*, *Stripe* entre outras ao invés de criar uma solução de pagamento sua e do zero.

Outra vantagem relevante é a atualização e manutenção. Quando um serviço externo atualiza sua API, geralmente oferece nova funcionalidades, correções de *bugs* ou melhorias de performance. Deste modo, os aplicativos que utilizam essa API podem se beneficiar dessas atualizações sem a necessidade de alterar o próprio código.

No entanto, a integração com APIs de terceiros também apresenta desafios. A disponibilidade é um fator crucial. Se o serviço externo enfrenta interrupções frequentes, isso pode afetar a experiência do usuário no aplicativo que depende dessa API. Além disso, questões de segurança, como a proteção dos dados transmitidos, são primordiais e devem ser tratadas com muita atenção (Rother, 2017).

No contexto de aplicativos móveis, as APIs são especialmente relevantes. Muitos apps dependem de dados em tempo real ou de recursos de plataformas externas. Como é o caso de aplicativos de previsão do tempo, que se integram a APIs meteorológicas, ou apps de redes sociais que utilizam APIs para publicar ou buscar informações de plataformas como *Twitter* "X", *Facebook*, *Instagram* entre outras.

Em nosso próximo exemplo, ilustramos a integração com uma API simples. Para tal, imagine um aplicativo que necessita mostrar informações sobre filmes, então ao invés de armazenar todos os dados sobre os mesmos localmente, podemos integrá-lo à API do *The Movie Database* <<https://developer.themoviedb.org/reference/intro/getting-started>> para obtermos detalhes atualizados sobre os filmes disponíveis.

Como em outras situações já descritas, antes de podermos testar nosso código será necessário instalarmos uma biblioteca com recursos específicos às nossas necessidades. Neste caso em particular, instalaremos o pacote **requests** utilizando o **pip** através do prompt de comando do *Windows*, onde deveremos executar a seguinte linha de comando:

```
pip install requests
```

A Figura 43 a seguir, ilustra o retorno de nosso SO frente ao que foi realizado.

```
Prompt de Comando
Microsoft Windows [versão 10.0.22621.2428]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\lincx>pip install requests
Requirement already satisfied: requests in
c:\users\lincx\appdata\local\programs\python\python311\lib\site-packages (2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2.0.0 in
c:\users\lincx\appdata\local\programs\python\python311\lib\site-packages (from requests)(3.2.0)
Requirement already satisfied: idna<4,>=2.5 in
c:\users\lincx\appdata\local\programs\python\python311\lib\site-packages (from requests)(3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in
c:\users\lincx\appdata\local\programs\python\python311\lib\site-packages (from requests)(2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in
c:\users\lincx\appdata\local\programs\python\python311\lib\site-packages (from requests)(2023.7.22)

[notice] A new release of pip is available: 23.2.1 -> 23.3
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\lincx>
```

**Figura 43:** Instalando o pacote **request** (com recursos que simplificam a comunicação na Internet) para uso de APIs

Em nosso exemplo ilustrado na Figura 44, utilizamos a biblioteca **requests** para fazer uma requisição à API do **TMDb**. O objetivo é buscar informações sobre um filme pelo título, então ao executarmos a função **buscar\_filme\_por\_titulo** a mesma imprimirá detalhe sobre o filme, como Nome: **nome**, Sinopse: **sinopse** e Data de Lançamento: **data\_lancamento**.

```
jogo.py > ...
1 import requests
2
3 # Defina a chave de API (obtenha a sua no site do TMDb)
4 API_KEY = "SUA_CHAVE_API"
5
6 BASE_URL = "https://api.themoviedb.org/3"
7
8 def buscar_filme_por_titulo(titulo):
9     url = f"{BASE_URL}/search/movie?api_key={API_KEY}&query={titulo}"
10    resposta = requests.get(url)
11
12    # Verificando se a resposta foi bem-sucedida
13    if resposta.status_code == 200:
14        dados = resposta.json()
15
16        # Extraiendo informações sobre o primeiro filme encontrado
17        if dados['results']:
18            primeiro_filme = dados['results'][0]
19            nome = primeiro_filme['title']
20            sinopse = primeiro_filme.get('overview', 'Sinopse não disponível')
21            # Usando get para evitar KeyError
22            data_lancamento = primeiro_filme.get('release_date', 'Data não disponível')
23            print(f"Nome: {nome}")
24            print(f"Sinopse: {sinopse}")
25            print(f"Data de Lançamento: {data_lancamento}")
26        else:
27            print("Filme não encontrado!")
28    else:
29        print(f"Erro ao acessar a API: {resposta.status_code}")
30
31 # Testando a função
32 buscar_filme_por_titulo("Inception")
33
```

**Figura 44:** Exemplo de integração com a API **TMDb** para obtenção de detalhes de filmes

Ao trabalhar com APIs, é sempre importante verificar a documentação do serviço. Nele você encontrará detalhes sobre os **endpoints**<sup>1</sup> disponíveis, os parâmetros requeridos e os seus possíveis retornos.

Em conclusão, a integração com APIs de terceiros enriquece os aplicativos, oferecendo uma gama de recursos e funcionalidades que seriam difíceis ou demoradas para serem desenvolvidas do zero. Entender as possibilidades e desafios da integração é fundamental para desenvolver aplicativos robustos, atualizados e relevantes em um cenário tecnológico em constante evolução.

Como vimos, muitos aplicativos precisam se comunicar com serviços externos para obter dados ou executar funções. Na Figura 45 a seguir, ilustramos um outro exemplo de utilização de API, agora da empresa **HG Weather** (<https://www.weatherapi.com/>), a qual nos possibilita obtermos informações de previsão do clima, para as mais diversas regiões do planeta.

```

1  from kivy.network.urlrequest import UrlRequest
2
3  API_URL = "https://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&query="
4
5  def fetch_weather(city):
6      req = UrlRequest(API_URL + city, on_success=on_success)
7
8  def on_success(req, result):
9      temp = result['current']['temp_c']
10     print(f'Temperatura em {city}: {temp}°C')

```

**Figura 45:** Exemplo de acesso à API de previsão do clima **weatherapi**

<sup>1</sup> São as localizações digitais expostas pela API a partir da API que recebe e responde às consultas. Cada endpoint é uma URL - Uniform Resource Locator que fornece a localização de um recurso no servidor da API.

## Desenvolvimento de Jogos para Smartphones

O desenvolvimento de jogos tem sido uma das áreas mais populares e inovadoras dentro da tecnologia, e o seu mercado para smartphones é um dos que mais cresce globalmente. Segundo a empresa de levantamento de dados sobre a área de games Newzoo (2021), os jogos móveis representaram 50% do mercado global de jogos naquele ano, com um valor de cerca de \$90 bilhões. Tais números, além de mostrarem a vitalidade econômica desse nicho, atestam sua relevância social, já que os smartphones estão presentes na vida de bilhões de pessoas.

Para ingressar nesse mundo fascinante, os desenvolvedores têm à disposição uma ampla variedade de ferramentas e frameworks. Entre elas destaca-se o nosso já conhecido **Kivy**.

Figurando entre os principais atrativos do framework, está a sua capacidade de processar e detectar gestos na tela. Com os smartphones sendo predominantemente controlados via toques, essa característica torna o Kivy uma ferramenta poderosa para o desenvolvimento de jogos. Em termos mais técnicos, o Kivy suporta mais de 20 diferentes tipos de gestos, incluindo movimentos de rotação, deslizamento e pinça (McGugan, 2016).

Vamos então considerar um exemplo prático para ilustrar a eficiência do Kivy para tal segmento. Imagine um jogo simples, onde o usuário tem que tocar em alvos que aparecem aleatoriamente na tela.

A Figura 46 a seguir, ilustra em detalhes o que queremos demonstrar.

```

jogo.py > ...
1   from kivy.app import App
2   from kivy.uix.button import Button
3   from random import randint
4
5   class GameApp(App):
6       def build(self):
7           # Criando um botão com o texto 'Toque em mim!'
8           btn = Button(text='Toque em mim!', on_release=self.move_button)
9           return btn
10
11   def move_button(self, btn):
12       # Repositionando o botão para um local aleatório após ser tocado
13       btn.pos = (randint(0, 300), randint(0, 400))
14
15   # Inicializando o jogo
16   GameApp().run()
17

```

PROBLEMS 1 OUTPUT TERMINAL PORTS DEBUG CONSOLE

```

[INFO] [GL] Using the "OpenGL" graphics system
[INFO] [GL] GLEW initialization succeeded
[INFO] [GL] Backend used <glew>
[INFO] [GL] OpenGL version <b'4.6.0 NVIDIA 537.42'>
[INFO] [GL] OpenGL vendor <b'NVIDIA Corporation'>
[INFO] [GL] OpenGL renderer <b'NVIDIA GeForce RTX 2060/PCIe/SSE2'>
[INFO] [GL] OpenGL parsed version: 4, 6
[INFO] [GL] Shading version <b'4.60 NVIDIA'>
[INFO] [GL] Texture max size <32768>
[INFO] [GL] Texture max units <32>
[INFO] [Window] auto add sdl2 input provider
[INFO] [Window] virtual keyboard not allowed, single mode, not docked
[INFO] [Base] Start application main loop
[INFO] [GL] NPOT texture support is available

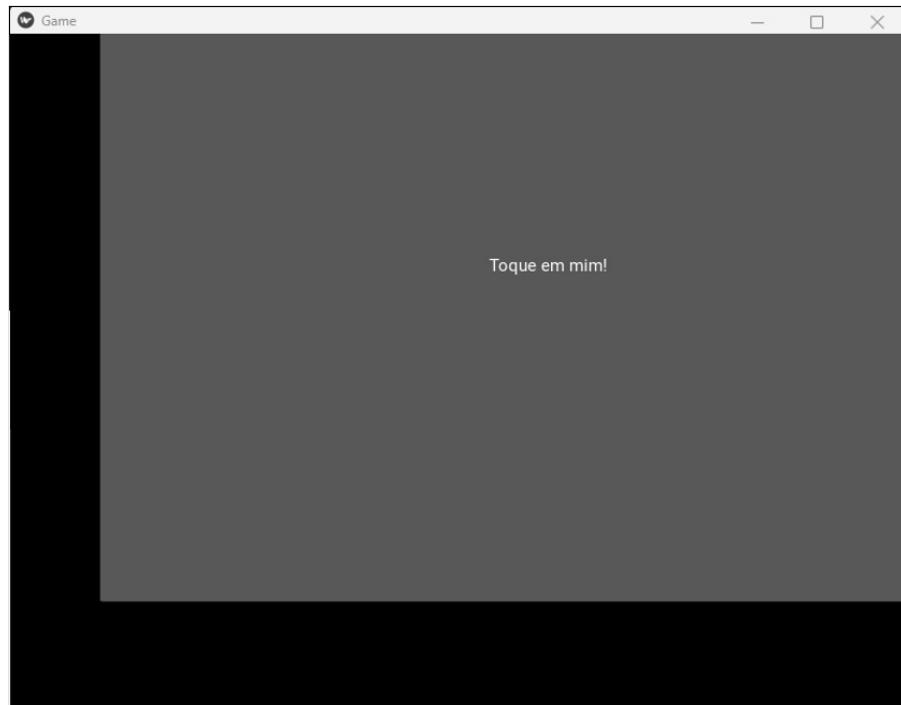
```

**Figura 46:** Exemplo de jogo em Python com avisos de saída em tempo de execução

Ao iniciar o jogo, um botão aparece na tela com o texto "**Toque em mim!**". Toda vez que o botão é tocado, ele se move para uma nova posição aleatória na tela.

O exemplo dado é bastante simples, mas imagine expandir isso para alvos múltiplos, diferentes níveis de dificuldade ou até mesmo adicionar recursos de som e gráficos avançados. O Kivy é capaz de suportar todas essas funcionalidades.

A Figura 47 a seguir, ilustra a aplicação com o botão em execução.



**Figura 47:** Botão do jogo em execução

Vamos então para um novo exemplo mais complexo e próximo da nossa realidade, um Jogo da Velha (*Tic Tac Toe*) simplificado. criá-lo, utilizaremos uma grade 3x3 e botões que podem ser tocados para registrar as jogadas.

```
jogo.py > ...
1  from kivy.app import App
2  from kivy.uix.gridlayout import GridLayout
3  from kivy.uix.button import Button
4
5  class TicTacToe(GridLayout):
6      def __init__(self, **kwargs):
7          super(TicTacToe, self).__init__(**kwargs)
8          self.cols = 3
9          for i in range(9):
10              btn = Button(text='', font_size=32)
11              btn.bind(on_press=self.player_move)
12              self.add_widget(btn)
13
14      def player_move(self, button):
15          if button.text == '':
16              button.text = 'X'
17          # Para este exemplo, usaremos 'X' para todas as jogadas
18  class TicTacToeApp(App):
19      def build(self):
20          return TicTacToe()
21
22  if __name__ == '__main__':
23      TicTacToeApp().run()
```

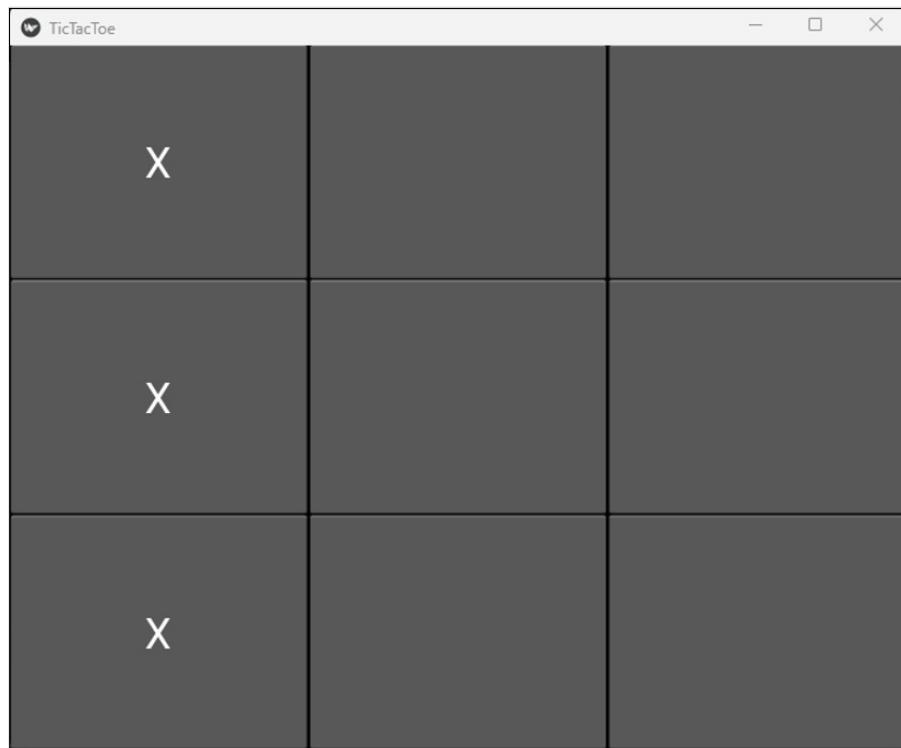
PROBLEMS 1 OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
[INFO] [GL] OpenGL renderer <b'NVIDIA GeForce RTX 2060/PCIe/SSE2'>
[INFO] [GL] OpenGL parsed version: 4, 6
[INFO] [GL] Shading version <b'4.60 NVIDIA'>
[INFO] [GL] Texture max size <32768>
[INFO] [GL] Texture max units <32>
[INFO] [Window] auto add sdl2 input provider
[INFO] [Window] virtual keyboard not allowed, single mode, not docked
[INFO] [Base] Start application main loop
[INFO] [GL] NPOT texture support is available
```

**Figura 48:** Exemplo de Jogo da Velha em Python com avisos de saída em tempo de execução

Em nosso exemplo ilustrado na Figura 48, criamos um *layout* de grade com 9 botões, representando o tabuleiro do jogo. Ao tocar um botão vazio, o texto do botão muda para 'X'. Claro, um jogo completo da velha também envolveria alternar entre 'X' e 'O' e verificar algum jogador ganhou, mas esse é apenas um começo!

A Figura 49 a seguir, ilustra o tabuleiro do jogo em execução.



**Figura 49:** Tabuleiro 3x3 do Jogo da Velha em execução

Continuando com a exploração das potencialidades oferecidas pelo *Kivy* para o desenvolvimento de jogos em dispositivos móveis, vamos construir agora um jogo de deslizar e mover, no qual o usuário pode arrastar um objeto pela tela.

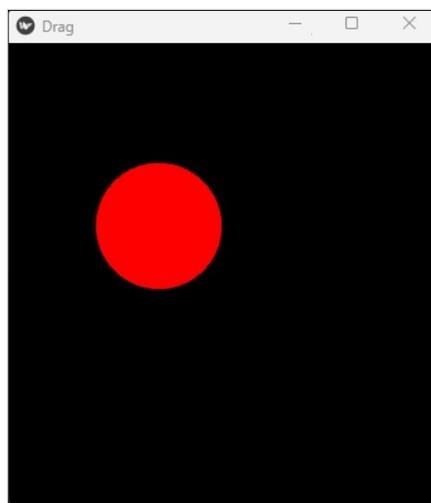
A Figura 50 a seguir, ilustra em detalhes o que queremos demonstrar.

```
jogo.py > ...
1  from kivy.app import App
2  from kivy.uix.widget import Widget
3  from kivy.uix.floatlayout import FloatLayout
4  from kivy.graphics import Ellipse, Color
5
6  class DraggableBall(Widget):
7      def __init__(self, **kwargs):
8          super(DraggableBall, self).__init__(**kwargs)
9          with self.canvas:
10              Color(1, 0, 0) # vermelho
11              Ellipse(pos=(self.center_x - 50, self.center_y - 50), size=(100, 100))
12              self.bind(pos=self.update_graphics_pos)
13              self.bind(size=self.update_graphics_size)
14
15      def update_graphics_pos(self, *args):
16          self.canvas.clear()
17          with self.canvas:
18              Color(1, 0, 0)
19              Ellipse(pos=(self.center_x - 50, self.center_y - 50), size=(100, 100))
20
21      def update_graphics_size(self, *args):
22          self.canvas.clear()
23          with self.canvas:
24              Color(1, 0, 0)
25              Ellipse(pos=(self.center_x - 50, self.center_y - 50), size=self.size)
26
27      def on_touch_down(self, touch):
28          if self.collide_point(*touch.pos):
29              self.pos = (touch.x - self.width / 2, touch.y - self.height / 2)
30              return True
31          return super(DraggableBall, self).on_touch_down(touch)
32
33      def on_touch_move(self, touch):
34          if self.collide_point(*touch.pos):
35              self.pos = (touch.x - self.width / 2, touch.y - self.height / 2)
36              return True
37          return super(DraggableBall, self).on_touch_move(touch)
38
39  class GameScreen(FloatLayout):
40      def __init__(self, **kwargs):
41          super(GameScreen, self).__init__(**kwargs)
42          ball = DraggableBall()
43          self.add_widget(ball)
44
45  class DragApp(App):
46      def build(self):
47          return GameScreen()
48
49  if __name__ == "__main__":
50      DragApp().run()
51
```

**Figura 50:** Exemplo de jogo deslizar e mover objeto

Neste caso, criamos um *widget* personalizado **DraggableBall** que representa um círculo. O método **on\_touch\_move** permite que o jogador arraste a bola (círculo) pela tela ao tocá-la.

A Figura 51 a seguir, ilustra a bola (círculo vermelho) que o usuário pode manusear.

**Figura 51:** Aplicativo de deslizar e mover objeto em execução

Estes são apenas dois exemplos básicos que utilizamos para demonstrar o que pode ser feito com o *Kivy*. Mas como já dissemos o framework oferece uma ampla variedade de *widgets* e funcionalidades que permitem criar jogos muito mais complexos, com a adição por exemplo, de imagens, sons, animações e integrações com outros sistemas.

Vale ressaltar que para jogos ou mesmo aplicativos mais complexos, por vezes teremos que utilizar novos recursos especializados até mesmo desenvolver soluções criativas de nossa própria autoria. Imagine uma situação onde um jogo possui uma ação que precisa executar tarefas tais como, disponibilizar ao usuário gráficos em altíssima resolução, interação do usuário com a rede (durante jogos multijogador) e que precisa sincronizar o som e os placares de todos os jogadores em tempo real, ou seja, em paralelo e simultaneamente.

Para que isso ocorra perfeitamente, veremos na próxima seção que será necessário o uso do conceito de *multithreading*, onde o processo será responsável por executar as tarefas descritas de forma independente e individual.

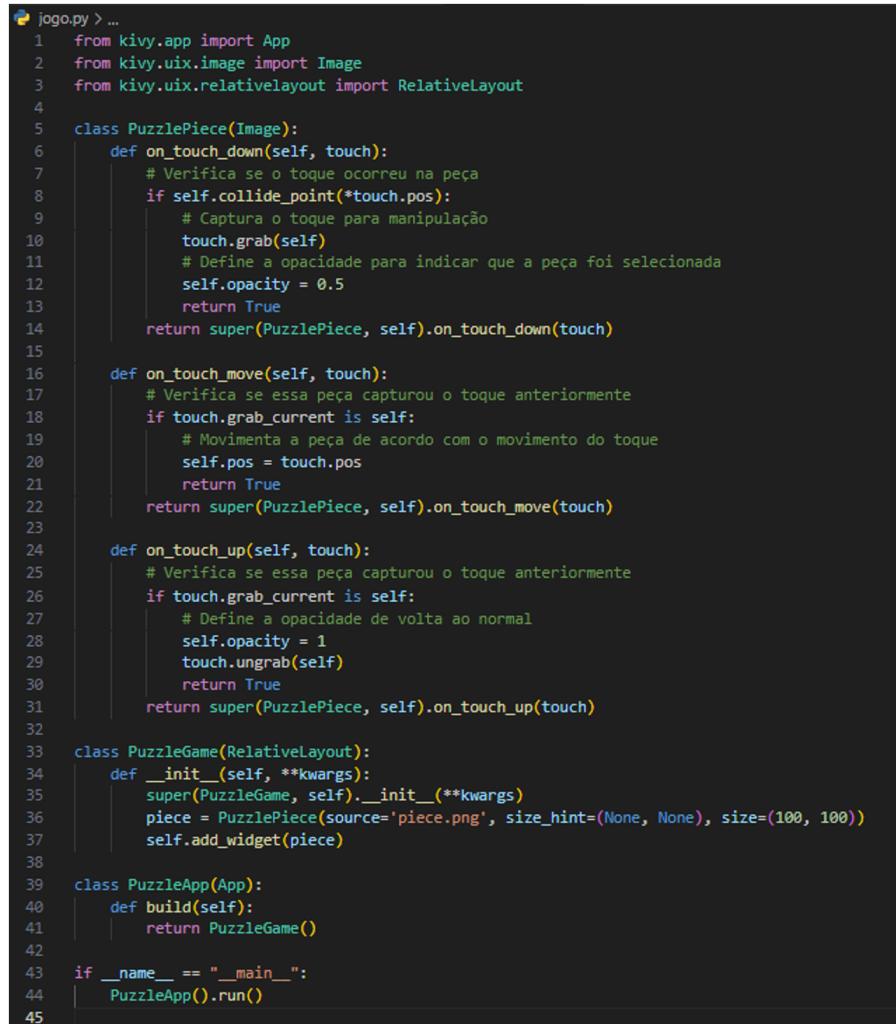
### Exemplo de Aplicação

Nos últimos anos, a mobilidade tornou-se uma das forças mais disruptivas no mundo da tecnologia. Com *smartphones* cada vez mais poderosos e a crescente importância dos aplicativos móveis, os desenvolvedores têm buscado constantemente por ferramentas eficazes para criar soluções atraentes e funcionais.

A interatividade é um pilar fundamental para a experiência do usuário em jogos. Ao pensar nisso, a capacidade do *Kivy* de interpretar gestos - desde os mais simples, como toques, até os mais complexos, como "pinça" ou "arrasto" - é uma característica valiosa (Smit, 2018). Pensando nisso, vamos considerar um novo cenário onde queremos criar um jogo de quebra-cabeça simples.

Neste jogo, os usuários precisarão arrastar peças até seus respectivos lugares no tabuleiro. A natureza intuitiva de arrastar e soltar as peças faz do *Kivy* uma escolha ideal para esse tipo de jogo (Jones, 2017).

O exemplo ilustrado na Figura 52, representa o esqueleto básico de um quebra-cabeça, onde a classe **PuzzlePiece** representa uma peça individual, enquanto a classe **PuzzleGame** representa o jogo em si.



```

jogo.py > ...
1  from kivy.app import App
2  from kivy.uix.image import Image
3  from kivy.uix.relativelayout import RelativeLayout
4
5  class PuzzlePiece(Image):
6      def on_touch_down(self, touch):
7          # Verifica se o toque ocorreu na peça
8          if self.collide_point(*touch.pos):
9              # Captura o toque para manipulação
10             touch.grab(self)
11             # Define a opacidade para indicar que a peça foi selecionada
12             self.opacity = 0.5
13             return True
14         return super(PuzzlePiece, self).on_touch_down(touch)
15
16     def on_touch_move(self, touch):
17         # Verifica se essa peça capturou o toque anteriormente
18         if touch.grab_current is self:
19             # Movimenta a peça de acordo com o movimento do toque
20             self.pos = touch.pos
21             return True
22         return super(PuzzlePiece, self).on_touch_move(touch)
23
24     def on_touch_up(self, touch):
25         # Verifica se essa peça capturou o toque anteriormente
26         if touch.grab_current is self:
27             # Define a opacidade de volta ao normal
28             self.opacity = 1
29             touch.ungrab(self)
30             return True
31         return super(PuzzlePiece, self).on_touch_up(touch)
32
33     class PuzzleGame(RelativeLayout):
34         def __init__(self, **kwargs):
35             super(PuzzleGame, self).__init__(**kwargs)
36             piece = PuzzlePiece(source='piece.png', size_hint=(None, None), size=(100, 100))
37             self.add_widget(piece)
38
39     class PuzzleApp(App):
40         def build(self):
41             return PuzzleGame()
42
43     if __name__ == "__main__":
44         PuzzleApp().run()
45

```

Figura 52: Exemplo de jogo de Quebra-Cabeça com *Kivy*

### Multithreading

O mundo moderno da computação se transformou significativamente com a capacidade de executar múltiplos processos simultaneamente. À medida que os aplicativos se tornam mais complexos e exigem maior responsividade, a habilidade de executar tarefas em paralelo tem se tornado crucial. Neste contexto, entra em cena o *multithreading*.

*Multithreading* refere-se à capacidade de uma unidade central de processamento (CPU) ou de um único núcleo em uma CPU de múltiplos núcleos gerenciar múltiplos processos (tarefas) ou *threads* (trilhas) simultaneamente (Silberschatz et al., 2020). *Python*, com linguagem de programação versátil e poderosa que é, oferece ferramentas para lidar eficientemente com *multithreading*.

Ao trabalhar com GUIs - *Graphical Interface User* ou interfaces gráficas de usuário, como em aplicativos desenvolvidos com *Kivy*, *multithreading* torna-se especialmente importante. Isso porque, enquanto uma *thread* pode estar ocupada processando uma tarefa pesada, uma outra *thread* pode manter a interface responsiva, garantindo assim uma boa experiência ao usuário (Fischer, 2018).

Usar *multithreading* em *Python*, especialmente com *Kivy*, pode parecer complexo à primeira vista, mas com um entendimento só dos conceitos, tal tarefa se torna mais simples. *Python* possui um módulo chamado **threading**, o qual oferece uma maneira de criar *threads*.

Mas por que isso é relevante para o desenvolvimento em *Kivy*? Imagine um aplicativo que baixe dados da Internet. Sem *multithreading*, o aplicativo poderia congelar ou parecer travado durante o *download* das informações, mas com *multithreading*, a interface do usuário continua responsiva e permitindo interações contínuas (Smith, 2018).

Vamos então a mais um exemplo prático. Nele implementamos um aplicativo para a simulação de um *download* com execução por *multithreading*.

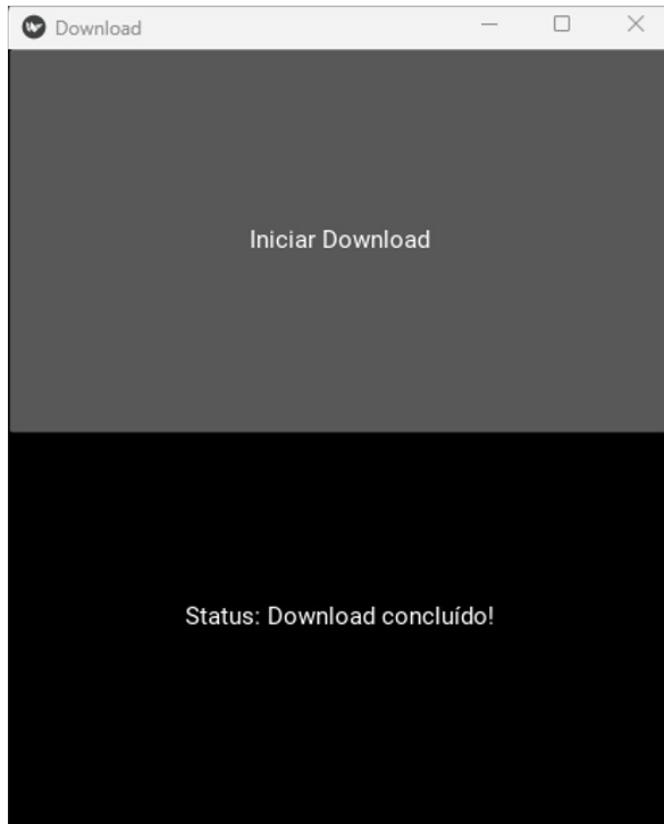
```
Download.py
1 import kivy
2 from kivy.app import App
3 from kivy.uix.button import Button
4 from kivy.uix.label import Label
5 from kivy.uix.boxlayout import BoxLayout
6 import threading
7 import time
8
9 kivy.require('2.0.0')
10
11 class DownloadApp(App):
12
13     def build(self):
14         self.b = BoxLayout(orientation ='vertical')
15
16         # Botão para iniciar o download
17         self.btn = Button(text ="Iniciar Download")
18         self.btn.bind(on_press = self.start_download)
19
20         # Etiqueta para mostrar o status do download
21         self.label = Label(text ="Status: Aguardando...")
22
23         self.b.add_widget(self.btn)
24         self.b.add_widget(self.label)
25
26         return self.b
27
28     def download_data(self):
29         time.sleep(5) # Simulando um download demorado
30         self.label.text = "Status: Download concluído!"
31
32     def start_download(self, instance):
33         self.label.text = "Status: Download iniciado..."
34         # Iniciando o download em uma nova thread
35         threading.Thread(target=self.download_data).start()
36
37 if __name__ == "__main__":
38     DownloadApp().run()
```

Figura 53: Exemplo de uso do recurso de *multithreading* com *Kivy*

Como ilustrado na Figura 53, nosso aplicativo possui um botão que, quando pressionado, inicia o *download* de dados. Tal operação é simulada com um *sleep* (espera) de 5 segundos, sendo que sem o uso de *multithreading* o aplicativo ficaria irresponsivo durante esse período.

esses 5 segundos, mas com a sua implementação o aplicativo permanece responsivo e o *status* do *download* é atualizado corretamente após a sua conclusão. Importamos ainda o pacote **time**, o qual fornece as classes para a manipulação de datas e horas.

A Figura 54 a seguir, ilustra o aplicativo de simulação de *download* em execução.



**Figura 54:** Aplicativo de simulação de *download* utilizando recurso de *multithreading* em execução

Este é um exemplo bastante simples da utilização de *threads*, mas a partir do mesmo você pode expandir o aplicativo para incluir exemplo, um calendário com notificações de alarme, integração com outras ferramentas e muito mais.

#### Saiba Mais

Como já trouxemos, a aplicação de *multithreading* no contexto computacional e por consequência para o desenvolvimento de aplicações modernas é bastante necessária. Todavia como já fizemos em outros tópicos de nosso curso, por questões de foco e tempo, não enveredaremos por todos os tópicos computacionais e/ou por todas as minúcias que permeiam diferentes situações. Neste sentido, dada a importância da compreensão do tema, indicamos como última sugestão do curso, que você assista ao vídeo intitulado "**Como usar Threads no Python**", o qual está disponível no YouTube o qual certamente lhe dará uma visão mais clarificada sobre a importância do tema e como aplicá-lo na prática.

#### Até Breve!

Caro(a) aprendiz, chegamos ao fim de nossa jornada pelo Curso de Desenvolvimento de Aplicativos Mobile. Cada módulo, cada de código e cada desafio enfrentado nos trouxe a este ponto. Você mergulhou profundamente no universo da programação móvel, explorando as nuances da criação de aplicativos com *Python* e *Kivy*.

Agora você está equipado(a) com o conhecimento e as ferramentas para fazer uma diferença real nesse cenário. No entanto, a aprendizagem não termina aqui, pois a tecnologia é um campo em constante evolução, e novas ferramentas, técnicas e paradigmas emergem a todo momento. Encorajamos você então a continuar sua jornada de aprendizado, seja explorando mais a fundo os tópicos abordados neste curso, seja se aventurando em novas áreas da tecnologia.

Lembrem-se de que cada desafio, cada erro e cada sucesso são partes integrantes do processo de crescimento. Persista, continue praticar e, o mais importante, continue a imaginar e a criar. O mundo dos aplicativos móveis é vasto e está cheio de oportunidade, esperando por desenvolvedores talentosos como você para moldá-lo.

Com gratidão e admiração por sua dedicação e esforço, desejamos a você muito sucesso em suas futuras empreitadas no mundo do desenvolvimento de aplicativos móveis. Até breve e boa sorte em todas as suas jornadas de codificação!

## Tags do conteúdo

Python, Mobile, Móvel, Smartphone, SO, Sistema Operacional

iOS, Android, Linux, Windows, Desktop

IDE, VSCode, Framework, Kivy, Xcode, Buildozer, Apps, Flutter

Variáveis, Tipos de Dados, Operador, Operadores, Estruturas de Controle

Estruturas Condicionais, Estruturas de Repetição, Funções, Atributos, Métodos

Programação Orientada a Objetos, POO, OOP

Classe, Encapsulamento, Herança, Polimorfismo, Pacote, Biblioteca

Interface de Usuário, Comportamento de Eventos, Widgets

Componentes Nativos, Layouts, Recursos de Design

KivyMD, Material Design, Guidelines, Componentes, Persistência, SQL, SQLite

Armazenamento Local, Armazenamento Remoto, Comunicação

Network, API, URL, HTTP, Push

UI, User Interface, Interface do Usuário, UX, User eXperience, Experiência do Usuário

Google Play, Apple App Store

Empacotamento, APK, IPA, requests, Jogos, Thread, Multithreading, threading

## Referências

AMADEO, R. *Android's 10th anniversary: A look back at its history*. Ars Technica, 2018. Disponível em: <<https://arstechnica.com>> Acesso em: 29 set. 2023.

AMADEO, R. *Android's Choice: Open Source Philosophy and Challenges*. Ars Technica, 2020. Disponível em: <<https://arstechnica.com/>> - Acesso em: 29 set. 2023.

APPLE. *App Store Review Guidelines*. 2021. Apple Inc., 2023. Disponível em: <<https://developer.apple.com/>> - Acesso em: 16 out.

CUEVAS, M. *The Apple Way: Understanding App Store's Success*. Forbes, 2021. Disponível em: <<https://www.forbes.com/>> - Acesso em: 13 out. 2023.

DEITEL, P. J. & DEITEL, H. M. *Python for Programmers: with Big Data and Artificial Intelligence Case Studies*. Pearson Illustration Edition, 2019.

FDCEa. *Tendências em UI e UX*. Fábrica de Conteúdos Educação, Editoração e Desenvolvimento de Sistemas Ltda - EPP, 2023.

FDCEb. *O processo de design*. Fábrica de Conteúdos Educação, Editoração e Desenvolvimento de Sistemas Ltda - EPP, 2023.

FINKEL, H. *Swift for the Really Impatient*. Addison-Wesley, 2020.

FISCHER, L. *Multithreading em Python: uma abordagem prática*. São Paulo: Edições Modernas, 2022.

FITZGERALD, B. *Software Testing: Ensuring Quality*. Wiley Publications, 2021.

GARTNER. *Gartner Says Worldwide Sales of Smartphones Recorded First Ever Decline During the Fourth Quarter of 2017*. - Egham, UK, February 22, 2018. Disponível em: <<https://www.gartner.com/en/newsroom/press-releases/2018-02-22-gartner-says-worldwide-sales-of-smartphones-recorded-first-ever-decline-during-the-fourth-quarter-of-2017>> - Acesso em: 28 set. 2023.

GOGGIN, G. *Global Mobile Media*. Routledge, 2011.

GOODRICH, M. T. et al.. *Data Structures and Algorithms in Python*. 2nd, John Wiley & Sons, 2019.

GOOGLE. *Developer's Guide to the Google Play Console*. 2020. Disponível em: <<https://developer.android.com/>> - Acesso em: 023.

JONES, M. *Harnessing Hardware with Kivy*. Mobile Development Journal, 2017.

KAHNEY, L. *Jony Ive: The Genius Behind Apple's Greatest Products*. Portfolio/Penguin, 2014.

KAY, Alan C. *The Early History of Smalltalk*. In: ACM SIGPLAN Notices. Association for Computing Machinery, New York, NY, USA n. 3, p. 69-95, mar. 1993.

LARDINOIS, F. *Google makes Kotlin a first-class language for writing Android apps*. TechCrunch, 2019. Disponível em: <<https://techcrunch.com/>> - Acesso em: 09 de out. 2023.

LEVIN, R. *Mac OS X and iOS Internals: To the Apple's Core*. Wiley, 2018.

LUTZ, Mark. *Learning Python*. O'Reilly Media, Inc., 2013.

LUTZ, Mark. *Programming Python*. 5th, O'Reilly Media, 2019.

MARTELLI, Alex et. al.. *Python Cookbook*. O'Reilly Media, 2005.

MARTIN, R. & MARTIN, M. *Mobile App Development with Modern IDEs*. Prentice Hall, 2020.

MCGUGAN, W. *Beginning Game Development with Python and Pygame: From Novice to Professional*. Apress, 2016.

MATTHES, E. *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. 2nd ed. [S.I.]: No Starch Press,

MEDNIEKS, Z.; DORNIN, L.; MEIKE, G.B.; NAKAMURA, M. *Programming Android: Java Programming for the New Generation Mobile Devices*. O'Reilly Media, Inc., 2015.

PHILLIPS, Dusty. *Python 3 Object-oriented Programming: Build robust and maintainable software with object-oriented design patterns in Python 3.8*. 3rd., Packt, 2018.

RAMALHO, Luciano. *Python Fluente: Programação Clara, Concisa e Eficaz*. Novatec, 2015.

ROSENBERG, A. & MATEESCU, M. *Data Privacy in Mobile Applications*. Stanford University Press, 2022.

ROTHER, Kristian. *Pro Python Best Practices: Debugging, Testing and Maintenance*. Apress, 2017.

SCHARFF, C. & VERMA, R. *Scrum to Support Mobile Application Development Projects in a Just-in-Time Learning Context*. Journal of Computing Sciences in Colleges, 26(5), 2011. p.255-261.

SEIFERT, D. *The Android Difference: How Google's Mobile OS Has Changed Over Time*. The Verge, 2019. Disponível em: <<https://www.theverge.com/>> - Acesso em: 04 out. 2023.

SILBERSCHATZ, A. et. al.. *Fundamentos de Sistemas Operacionais: Princípios básicos*. Rio de Janeiro: LTC, 2020.

SLATKIN, Brett. *Effective Python: 90 Specific Ways to Write Better Python*. 2nd, Addison-Wesley Professional, 2019.

SOMMERVILLE, I. *Software Engineering*. 11th ed., Addison-Wesley, 2021.

SMITH, R. *Desenvolvimento Ágil com Kivy e Python*. Belo Horizonte: Editora Futura, 2021.

SMITH, A. *Responsive Apps with Kivy: Asynchronous Patterns*. O'Reilly Media, 2018.

STATCOUNTERa. *Mobile Operating System Market Share Worldwide – Jan-2009 - Set-2023*. StatCounter®, 2023. Disponível em: <<https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-200901-202309>> - Acesso em: 07 de out. 2023.

STATCOUNTERb. *Mobile Operating System Market Share Brazil – Jan-2009 - Set-2023*. StatCounter®, 2023. Disponível em: <<https://gs.statcounter.com/os-market-share/mobile/brazil/#monthly-200901-202309>> - Acesso em: 07 de out. 2023.

SWEIGART, Al. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. 2nd, San Francisco: No St Press, 2019.

VAN ROSSUM, G. & DRAKE F. L. *An Introduction to Python*. Network Theory Ltd., 2011.

WEST, J. & MACE, M. *Browsing as the killer app: Explaining the rapid success of Apple's iPhone*. Telecommunications Policy, 4), 2007. p.270-286.

#### Nota:

Todas as marcas citadas e/ou exibidas neste material, pertencem aos seus respectivos fabricantes e/ou desenvolvedores.