

PYTHON PARA TODOS

DA INTRODUÇÃO À PRÁTICA

Aprenda Python com exemplos práticos,
aplicações reais e exercícios passo a passo.





▶▶▶▶ O QUE VOCÊ IRÁ APRENDER

Um guia amigável para iniciar sua jornada no mundo da programação com Python.



Conteúdo.py

- O que é Python
- Primeiros passos na linguagem
- Principais tipos de dados do Python
- Aprofundando-se com Strings
- F-Strings
- Operadores Matemáticos
- Conversão de tipagem
- Funções embutidas (built-in)
- Estruturas condicionais
- Praticando o que você aprendeu
- Laços de repetição (Loops)
- Funções
- Coleções de dados
 - Listas
 - Tuplas
 - Dicionários
- Tratamento de erros





PROGRAMADOR AVENTUREIRO

Mais de **11 mil** alunos

Alunos de mais de **60 países**

Plataforma especializada em **Python** e suas tecnologias

Mais de **40 cursos** e **700 horas** de conteúdo

EMPRESAS ONDE NOSSOS ALUNOS TRABALHAM



PROGRAMADORAVENTUREIRO.COM



AVISO

Você está prestes a dar o próximo grande passo na sua jornada como desenvolvedor Python.

Nas suas mãos está um guia completo para dominar as principais habilidades de Python e construir uma carreira promissora. Mas seus resultados podem ser ainda melhores com um plano de estudos bem estruturado.

Para entender como se tornar um Desenvolvedor Full Stack Python em tempo recorde com o cronograma mais eficiente do mercado, assista à aula gratuita que preparei para você.

CLIQUE PARA ASSISTIR A AULA

(ou acesse programadoraventureiro.com/pro)

O QUE VOCÊ VAI APRENDER NA AULA


- Como é possível se tornar um programador Python profissional em menos de 3 meses, mesmo começando do zero.
- O cronograma passo a passo para se especializar em Python.
- Pré-requisitos essenciais para começar na programação e se destacar no mercado.
- Como ganhar pelo menos R\$ 5.000 por mês como desenvolvedor Python.
- Como ganhar até R\$ 6.500 na área de dados.
- Estratégias para aprender e ganhar dinheiro enquanto estuda programação.
- Como lidar com a falta de experiência em programação ou pouco tempo para estudar.
- As etapas essenciais para dominar Python e desenvolver projetos práticos.
- Como superar os obstáculos mais comuns no seu aprendizado.
- Dicas para criar um portfólio atraente e competitivo.

PROGRAMADORAVENTUREIRO.COM



SOBRE O AUTOR

Olá, sou Dalton Peixoto, especialista em tecnologia e educação, com mais de 15 anos de experiência em desenvolvimento de software e ciência de dados. Sou o fundador do **Programador Aventureiro**, uma escola online de programação e ciência de dados que tem ajudado milhares de pessoas ao redor do mundo a darem seus primeiros passos ou a avançarem em suas carreiras na área de tecnologia.



Ao longo da minha jornada profissional, tive a oportunidade de desenvolver mais de 12 modelos de inteligência artificial no setor financeiro, aplicando linguagens como Python, JavaScript, PHP e Java para resolver problemas desafiadores e promover inovação. Essa experiência técnica me deu a base para ensinar com confiança e oferecer aos meus alunos uma formação sólida e prática.

Tenho paixão por transformar vidas através da educação. Nos últimos seis anos, ajudei mais de **11 mil pessoas** a entrarem para o mercado de tecnologia e conquistarem salários mais altos, além de ensinar milhares de pessoas a aprenderem programação do zero, seja pelas redes sociais ou pelo YouTube. **Ver meus alunos alcançando seus objetivos é o que me motiva todos os dias.**

Com uma abordagem prática, inclusiva e focada em resultados, meus cursos foram avaliados com uma média de 4,8/5, reflexo do meu compromisso em oferecer conteúdos de alta qualidade. Acredito que qualquer pessoa pode aprender a programar, independentemente da experiência prévia, e estou aqui para garantir que essa jornada seja acessível, eficiente e transformadora.

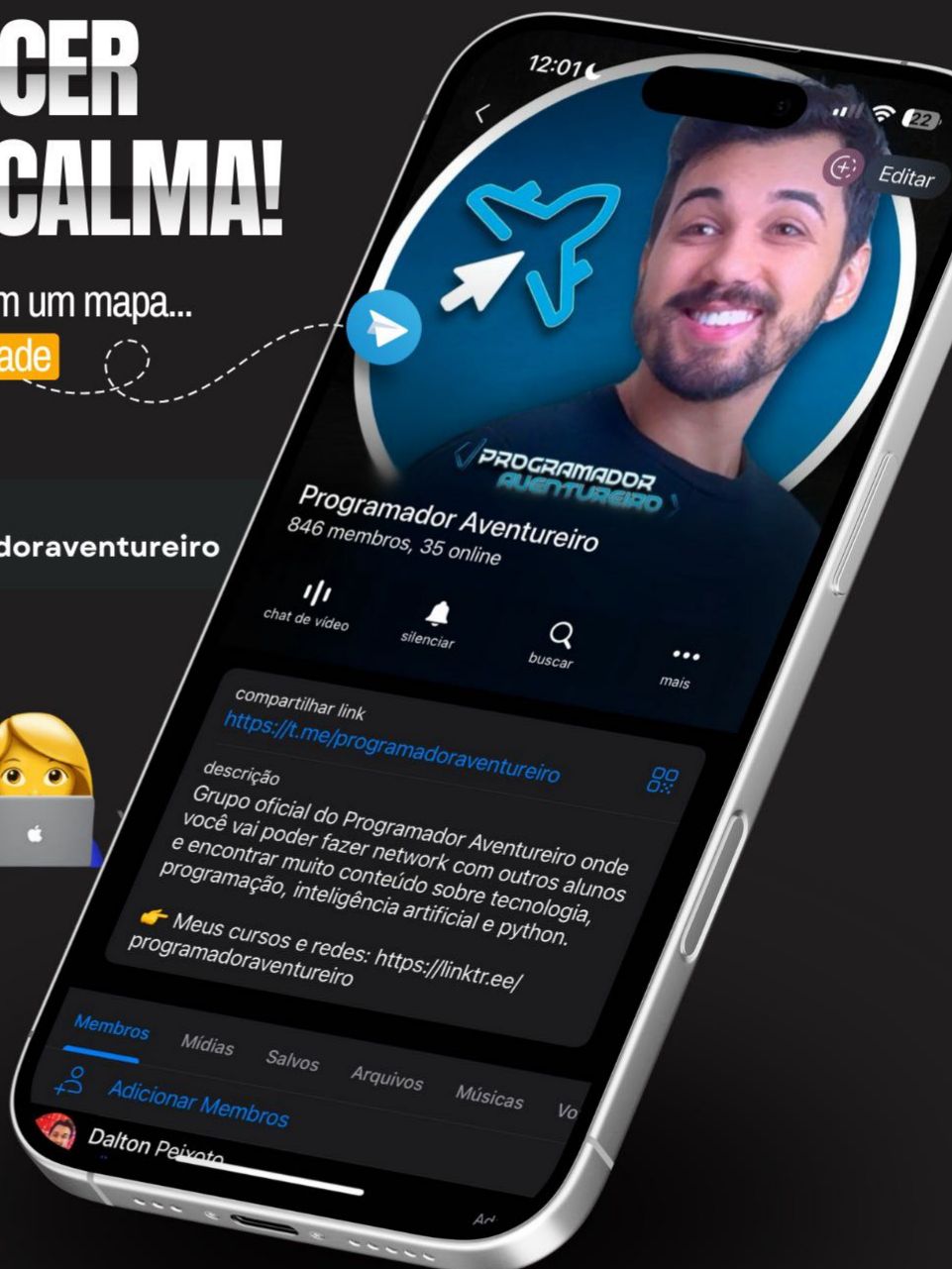
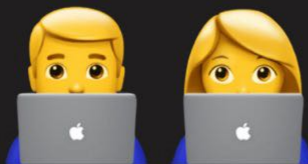
O futuro pertence a quem se prepara hoje

SE APARECER UM BUG, CALMA!

Todo bom aventureiro tem um mapa...
....e o nosso é a **comunidade**



Entre agora pelo link
t.me/programadoraventureiro



Eu sei que começar a aprender algo novo é bastante desafiador e é inevitável que surjam dúvidas pelo caminho, é claro que eu estou sempre aqui pra te ajudar, mas sabia que existe uma comunidade de alunos onde todos compartilham suas dificuldades, novos projetos que estão desenvolvendo e até mesmo vagas de emprego para uma determinada tecnologia?

E você pode fazer parte dessa comunidade, basta entrar no grupo do Telegram de alunos do Programador Aventureiro

CLIQUE AQUI PARA ENTRAR

(ou acesse <https://t.me/programadoraventureiro>)

O que é Python

Python é uma linguagem de programação de alto nível (High Level Language) totalmente focada no desenvolvedor, o que significa que ela se assemelha muito com a linguagem humana e prioriza o esforço do homem perante a máquina.

Desde sua origem essa linguagem foi pensada e construída para ser simples e com sintaxe de fácil compreensão. O que foi primordial para dar a ela grande popularidade entre profissionais de diversas áreas além do TI, como matemáticos, cientistas, engenheiros, pesquisadores, entre outros.

Open source, isto é, seu código fonte é aberto possibilitando que qualquer pessoa possa desenvolver melhorias e novas aplicações para a linguagem.

E, multiparadigma o que significa que o Python suporta diversos paradigmas de desenvolvimento, como programação funcional, imperativa, interpretada, orientada a objetos, entre outros. Assim um mesmo programa pode ser feito utilizando diferentes paradigmas.

Onde tudo começou

Seu início se deu no início dos anos 90 com o matemático holandês Guido Van Rossum. Guido era pesquisador e precisava otimizar outro software que trabalhava na época chamado ABC.



A idéia era torná-lo mais ágil e melhorar sua performance e interação com o sistema operacional, reduzindo o tempo total de execução dos programas ganhando mais eficiência no conjunto.

Com isso, era preciso que a nova linguagem capaz de trazer essas melhorias fosse descomplicada e flexível para que fosse possível a construção de scripts desde o mais simples até sistemas completos e extremamente poderosos. Assim surgiu o Python !

Essa proposta de ser simples desde a sua concepção garantiu ao Python uma grande aderência da comunidade científica contando hoje além dos desenvolvedores de software, também com contadores, biólogos, matemáticos, físicos e profissionais de diversas outras áreas.

Aplicações da linguagem Python

Por ser extremamente versátil e totalmente modularizado o Python permitiu que fosse incorporado diversas funcionalidades a linguagem tornando-a cada vez mais potente.

Tudo isso fez com que diversas empresas adotassem a linguagem em suas aplicações como Google, YouTube, Facebook, NASA (isso mesmo!), entre outras.



Os módulos, bem como o suporte a orientação de objetos, ampliam (e muito) a capacidade da linguagem, acrescentando recursos diferentes e implementando novas funcionalidades. Dentre elas temos:

- **Django:** é um framework muito poderoso utilizado na construção de ferramentas web, web services e páginas completas
- **ZODB:** utilizada para gerenciamento de banco de dados
- **Pygame:** extensão aplicada ao desenvolvimento de jogos eletrônicos

E muitos outros módulos, bibliotecas e frameworks que podem ser encontrados no [PyPi](#) (Python Package Index) que é um repositório de software para a linguagem de programação Python.

O que mais pode ser feito com Python?

Como falamos até aqui, o Python é muito utilizado em diversas áreas, principalmente nas relacionadas a tecnologia e análise de dados, pesquisa, inteligência artificial, machine learning e desenvolvimento de algoritmos.

Abaixo listei algumas utilizações dele em cada uma destas áreas:

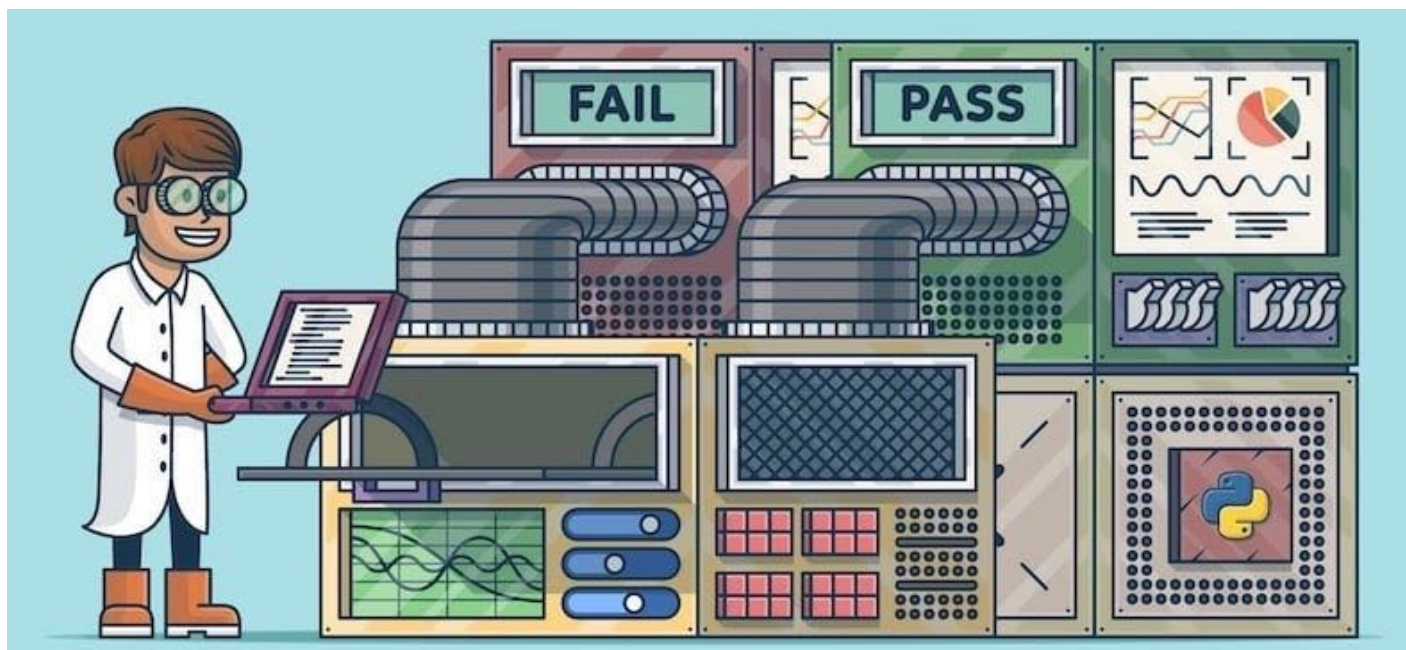
Automação de rotinas



Automatizar tarefas além de economizar tempo também elimina boa parte dos erros operacionais envolvidos.

Graças a diversas bibliotecas nativas da linguagem que estão disponíveis para o desenvolvedor juntamente com a instalação do Python, esse tipo de automação se torna fácil e intuitivo. Devido a essas bibliotecas dizemos que o Python é uma linguagem que possui “baterias incluídas”.

TDD e Testes automatizados



Fazer o enquadramento de testes é muito importante para qualquer programa, com ele conseguimos validar que nosso código continuará funcionando da mesma forma ou pelo menos com o mesmo resultado esperado.

Em algumas linguagens essa tarefa é por vezes chata e verbosa mas com o Python diversas estruturas de testes foram integradas ao seu código fonte além de possuir módulos específicos para a realização desses enquadramentos de testes.

Isso é mais uma vantagem da linguagem e a razão pela qual desenvolvedores de software em Python utilizam muito TDD (Test Driven Development), fazendo que suas aplicações sejam muito mais robustas e a prova de falhas.

Desenvolvimento WEB



Uma das funcionalidades mais comuns é o desenvolvimento de aplicações WEB, desde sites simples, landing pages, hotpages e blogs até ERPs completas que fazem a administração de empresas, realizam vendas de produtos e serviços, web services que integram recursos de API, bancos de dados e geram valor para o negócio.

A variedade é grande e o desenvolvedor Python que pretende desenvolver algum tipo de plataforma web conta com diversos frameworks que facilitam (e muito) essa empreitada. Dentre os mais comuns e famosos temos o [Django](#), [Flask](#) e [FastAPI](#).

Big Data



Big Data é a área de conhecimento que trabalha com grandes volumes de dados (por isso seu nome), e é onde o Python mais se destaca como sendo um facilitador uma vez que possui sintaxe de fácil compreensão fazendo que a curva de aprendizado seja relativamente pequena, até por profissionais que não estão diretamente ligados a áreas de programação.

Como comentei anteriormente ele possui diversas “baterias incluídas” o que facilitam a criação de scripts já que podemos implementar diversas funcionalidades aos nossos códigos sem a necessidade de instalar programas ou bibliotecas de terceiros.

Com ele conseguimos analisar, exibir e processar dados com eficiência, rapidez e clareza.

Algumas das bibliotecas mais conhecidas para se trabalhar com o Big Data são [Pandas](#) (análise e tratamento de dados), [Matplotlib](#) (visualização e exibição de dados), [Numpy](#) (tratamento de dados) e [PySpark](#) (análise e tratamento de dados).

Ciência de dados



Quase sempre quem houve falar de ciência de dados também ouve bastante de Python ou R. É notório que o Python ganhou muito destaque nessa área e apesar de não ser a única usada nesse campo da ciência com certeza é a mais querida e usada tanto no Brasil quanto internacionalmente.

Um dos motivos disso é a vasta gama de exemplos e soluções disponibilizadas pela comunidade pythônica ao redor do mundo, isso faz com que diminua consideravelmente as chances de um programador precisar lidar com um problema sem solução aparente.

A comunidade brasileira é grande e muito ativa, o que é mais um ponto positivo para nós 😊

IA (Inteligência Artificial)



Dando continuidade as aplicações mais comuns da linguagem e ao tema de ciência de dados temos a inteligência artificial, muito associada aos conceitos de machine learning (aprendizado de máquina).

Esse recurso é utilizado por diversas empresas normalmente com intuito de prever alguma característica, gosto ou intenção do usuário, aprendendo com seus costumes e hábitos para gerar resultados ou soluções personalizados para cada indivíduo.

Por exemplo, quando você assiste a um filme na Netflix é incrível como depois a plataforma te indica um amplo catálogo de filmes correlacionados com seus interesses e gostos.

Isso é feito com machine learning, a “máquina” aprende os seus gostos, preferências e hábitos e te indica filmes que possam te interessar.

Outro exemplo é o Google, mas dessa vez ele utiliza seus hábitos de pesquisa para prever a sua intenção de busca trazendo como resultados exatamente a resposta que você procurava.

Essa é uma área em grande expansão e com um leque gigantesco de possíveis

utilizações.

Entre as bibliotecas mais comuns voltadas ao aprendizado de máquina estão o [TensorFlow](#), [PyThorch](#), [Keras](#) e outras.

Computação gráfica



Na área da computação gráfica também temos muitas ferramentas e aplicações utilizando Python.

Existem bibliotecas e pacotes específicos para essa finalidade como o [PyOpenGL](#) e o [PyGame](#) mas também programas inteiros de computação gráfica que utilizam Python como sua linguagem principal, é o caso do Blender, um poderoso software de modelagem 3D gratuito que foi usado na criação dos curta metragens Sintel e Big Buck Bunny além do longa **Next Gen compada pela Netflix por \$ 30 milhões de dólares.**

Outros filmes que o Python também esteve envolvido na criação foram os novos da saga **Star Wars**, todos os seus efeitos gráficos foram produzidos pela Industrial Light & Magic.

Principais vantagens do Python



Como já deu pra perceber até aqui as vantagens de aprender e se dedicar a linguagem Python são muuuuittas e aqui vai mais uma: **os profissionais especializados nessa linguagem estão em falta no mercado!**

Ou seja, aprender essa linguagem será um **GRANDE DIFERENCIAL** na sua carreira e sua concorrência será baixíssima!

Alguns outros benefícios do Python são:

Simplicidade e facilidade de aprendizado

Dizemos que a curva de aprendizado de algum assunto é considerada baixa quando algo possui fácil compreensão e assimilação do estudante e, quanto mais difícil e denso for esse assunto maior será sua curva de aprendizado, isto é, mais esforço, empenho e dedicação o estudante terá que ter para entender o conteúdo.

A curva de aprendizagem do Python é relativamente baixa, sendo assim podemos dizer que é uma linguagem que o aluno consegue absorver rapidamente os conceitos e conteúdos.

Multiplataforma, multiparadigma e extensível

A capacidade de ser adaptativo é um grande diferencial e destaque do Python, a linguagem pode ser escrita sob diversos paradigmas desde os mais simples como o imperativo até os mais complicados como o orientado a objetos que é um paradigma considerado para "experts".

Também é suportado por múltiplas plataformas e sistemas operacionais o que o torna muito versátil e praticamente sem barreiras.

Além disso, o Python possui propriedades extensíveis podendo ser agregado a ele muitas funcionalidades desenvolvidas por terceiros para os mais diversos propósitos, tendo à sua disposição mais de 125.000 (cento e vinte e cinco mil) bibliotecas!

Licença de uso público

Resumidamente isso significa que o Python é uma linguagem Open Source, de código aberto, e totalmente gratuito, seja pra uso, pessoal ou comercial. Você pode desenvolver qualquer aplicação usando a linguagem e até mesmo vender ela e não precisará pagar um centavo de direitos autorais por isso.

E tem mais, a maior pesquisa realizada na área da programação, a [StackOverflow Survey](#), realizada em 2020 no mundo inteiro perguntou aos desenvolvedores qual linguagem eles mais gostavam de programar e adivinhe: **Python ficou em 1º lugar!**

Sintaxe do Python

Sua sintaxe é das mais simples e de fácil compreensão entre todas as linguagens programação atuais. Dentre suas características mais marcantes temos:

- Não utiliza ponto e vírgula (;) para finalizar uma instrução
- Utiliza indentação por espaços
- Uma variável pode armazenar diferentes tipos de dados
- Não há chaves ({}) para delimitar o início e final de um bloco de código



```
numero1 = 5
```

```
numero2 = 8
```

```
resultado = numero1 + numero2
```

```
print('O resultado é:')
```

```
print(resultado)
```

Primeiros passos

Como você viu no capítulo anterior o Python é uma linguagem de programação verdadeiramente versátil, amada tanto por desenvolvedores web, cientistas de dados e engenheiros de software. E há várias boas razões para isso!

- Python é de código aberto e possui uma ótima comunidade de suporte.
- Além disso, possui bibliotecas de suporte extensivas.
- Suas estruturas de dados são amigáveis ao usuário.

Uma vez que você pegue o jeito, sua velocidade de desenvolvimento e produtividade irão disparar!

A maioria dos computadores com Windows e Mac já vem com Python pré-instalado. Você pode verificar isso através de uma pesquisa na Linha de Comando:

```
python -V
```

Esse comando irá retornar a versão do Python instalada no seu computador. Caso retorne um erro é provável que o Python não esteja instalado, então será necessário fazer o download dele pelo site oficial: <https://www.python.org/downloads/>

O apelo particular do Python é que você pode escrever um programa em qualquer editor de texto, salvá-lo no formato .py e depois executá-lo via Linha de Comando. Mas, à medida que você aprende a escrever códigos mais complexos ou se aventurar na ciência de dados, pode querer mudar para um IDE ou IDLE.

O que é IDLE (Ambiente de Desenvolvimento e Aprendizagem)

O IDLE vem com cada instalação do Python. Sua vantagem sobre outros editores de texto é que ele destaca palavras-chave importantes (por exemplo, funções de `string`), tornando mais fácil para você interpretar o código. O Shell é o modo padrão de operação para o IDLE do Python. Em essência, é um loop simples que realiza os seguintes quatro passos:

- Lê a instrução Python
- Avalia os resultados
- Imprime o resultado na tela
- E então retorna para ler a próxima instrução.

O shell do Python é um ótimo lugar para testar pequenos trechos de código. Para habilitar o Shell do Python, você pode simplesmente abrir o IDLE após a instalação do Python. A janela do Shell aparecerá automaticamente, permitindo que você comece a digitar seus comandos e scripts imediatamente.

No entanto, à medida que seus projetos se tornam mais complexos, você pode querer explorar outras ferramentas que oferecem mais funcionalidades e uma experiência de desenvolvimento mais rica. Discutiremos sobre algumas opções de IDE e editores de código nas próximas páginas.

Visual Studio Code (VSCode)

O **VSCode** é um editor de código leve e poderoso, amplamente utilizado por desenvolvedores de Python e outras linguagens. Algumas de suas vantagens incluem:

- **Suporte a extensões:** Você pode adicionar funcionalidades com milhares de extensões disponíveis, incluindo suporte para Python.
- **IntelliSense:** Oferece autocompletar e dicas de código, o que acelera o desenvolvimento.
- **Integração de terminal:** Você pode executar seus scripts diretamente do terminal embutido.
- **Controle de versão:** Facilita o uso de sistemas como Git para versionamento de código.

Como Instalar o VSCode

1. Baixar e Instalar:

- Acesse o site oficial: <https://code.visualstudio.com/>.
- Baixe o instalador adequado para seu sistema operacional (Windows, macOS ou Linux).
- Siga as instruções do instalador para concluir a instalação.

2. Instalar a Extensão do Python:

- Abra o VSCode.
- Vá para a aba de Extensões (ícone de quadrado dividido na barra lateral esquerda ou pressione **Ctrl + Shift + X** no Windows/Linux ou **Cmd + Shift + X** no MacOS).
- Na barra de pesquisa, digite **Python**.
- Encontre a extensão oficial da Microsoft chamada **Python** e clique em **Instalar**.

JupyterLab

O **JupyterLab** é uma interface de desenvolvimento interativa baseada em web, que é especialmente útil para data science e aprendizado de máquina (machine learning). Algumas de suas vantagens incluem:

- **Documentação interativa:** Permite que você misture código, visualizações e documentação em um único documento, facilitando a apresentação de resultados.
- **Suporte a múltiplas linguagens:** Embora seja mais conhecido por Python, o Jupyter suporta diversas linguagens de programação.
- **Visualização de dados:** Integra facilmente bibliotecas de visualização, como Matplotlib e Seaborn.

Como Instalar o JupyterLab

1. Instalação via pip:

- Abra o terminal ou prompt de comando.
- Execute o seguinte comando para instalar o JupyterLab:

```
pip install jupyterlab
```

2. Iniciar o JupyterLab:

- Após a instalação, você pode iniciar o JupyterLab executando:

```
jupyter lab
```

- Isso abrirá uma nova aba no seu navegador, onde você pode criar e gerenciar seus notebooks.

Google Colab

O **Google Colab** é uma alternativa ao JupyterLab que roda em nuvem, permitindo que você execute notebooks Python sem a necessidade de instalação local. Algumas de suas vantagens incluem:

- **Acesso fácil:** Como é baseado em nuvem, você pode acessar seus notebooks de qualquer lugar e em qualquer dispositivo com internet.
- **Recursos de computação:** O Colab oferece GPUs e TPUs gratuitas, tornando-o ideal para treinamento de modelos de aprendizado de máquina.
- **Integração com Google Drive:** Salva automaticamente seus notebooks no Google Drive, facilitando o compartilhamento e a colaboração.

Como Usar o Google Colab

1. Acessar o Google Colab:

- Vá para o site oficial: <https://colab.research.google.com/>.
- Faça login com sua conta do Google para permitir o salvamento dos scripts no seu Google Drive.

2. Criar um Novo Notebook:

- Clique em **File** (Arquivo) > **New Notebook** (Novo Notebook).
- Você pode começar a escrever seu código Python diretamente nas células do notebook.

Principais Tipos de Dados no Python

Em Python, tudo é considerado um "objeto", e cada objeto possui um tipo de dado específico. Esses tipos de dados determinam como os objetos são armazenados na memória e as operações que podem ser realizadas sobre eles. Vamos explorar os mais comuns:

1. Inteiros (int)

Os números inteiros representam valores positivos ou negativos sem casas decimais. Eles são amplamente usados para operações matemáticas básicas e contagem.

Exemplos:

```
numero_positivo = 10  
numero_negativo = -7  
zero = 0
```

2. Números de Ponto Flutuante (float)

Os números de ponto flutuante são usados para representar valores com casas decimais, como 3.14 ou -0.001. Eles são essenciais em cálculos que exigem precisão.

Exemplos:

```
pi = 3.14  
temperatura = -12.5  
altura = 1.75
```

3. Strings (str)

Strings são cadeias de caracteres, usadas para armazenar texto. Elas são delimitadas por aspas simples (') ou duplas ("). Em Python, as strings são **imutáveis**, ou seja, não podem ser modificadas diretamente após serem criadas.

Exemplos:

```
cumprimento = "Olá, mundo!"  
nome = 'Python'
```

4. Booleanos (bool)

O tipo booleano é usado para representar os valores **True** (verdadeiro) ou **False** (falso). É frequentemente utilizado em condições e comparações.

Exemplos:

```
maior_de_idade = True  
chovendo = False  
  
# Comparações que retornam booleanos  
idade = 18  
print(idade > 18) # Saída: False  
print(idade == 18) # Saída: True
```

Os booleanos são a base para expressões condicionais, como **if** e **while**, que veremos mais adiante.

5. Coleções de Dados

Além dos tipos simples acima, Python possui tipos que permitem trabalhar com coleções de dados. Estes são fundamentais para organizar e processar grandes

volumes de informações.

- **Listas (list):** São coleções ordenadas e mutáveis, capazes de armazenar vários tipos de dados.
- **Dicionários (dict):** Armazenam pares de chave e valor, permitindo acesso rápido a dados através de suas chaves.
- **Tuplas (tuple):** Semelhantes às listas, mas imutáveis, ou seja, não podem ser alteradas após sua criação.

Exemplos de Coleções:

```
# Lista
frutas = ["maçã", "banana", "laranja"]

# Dicionário
aluno = {"nome": "João", "idade": 20, "curso": "Engenharia"}

# Tupla
coordenadas = (10, 20)
```

Essas estruturas serão abordadas em detalhes nos próximos capítulos, onde aprenderemos como manipulá-las para resolver problemas do mundo real.

Os tipos de dados apresentados aqui são a base da programação em Python. Saber como usá-los e entendê-los é essencial para qualquer aplicação, seja em cálculos matemáticos, manipulação de textos ou organização de dados. Nos próximos capítulos, exploraremos as coleções de dados em maior profundidade, além de outras ferramentas que tornam Python tão poderoso e versátil.

Aprofundando-se com Strings

As strings são um dos tipos de dados mais importantes e amplamente usados no Python. Elas representam sequências de caracteres e são usadas para lidar com qualquer tipo de texto. Seja para exibir mensagens ao usuário, armazenar informações textuais, manipular nomes, endereços, mensagens ou arquivos, as strings estão presentes em quase todos os programas.

No Python, qualquer sequência de caracteres delimitada por aspas (simples, duplas ou triplas) é considerada uma string. Mesmo números, se escritos entre aspas, são interpretados como texto. Por exemplo, `"123"` é uma string, enquanto `123` é um número inteiro.

O que é uma String?

Uma string é, essencialmente, uma sequência ordenada de caracteres, que pode incluir letras, números, símbolos e até espaços em branco. No Python, tudo que for tratado como texto será considerado uma string.

Para que Servem as Strings?

Strings são úteis para:

- Armazenar mensagens.
- Manipular texto (substituir palavras, juntar frases, dividir sentenças).
- Lidar com dados de entrada fornecidos por usuários.
- Exibir resultados para o usuário final.

Como o Python Enxerga Strings?

O Python trata strings como coleções de caracteres, cada um com uma posição (ou índice). Por isso, é possível acessar partes específicas de uma string usando sua posição, manipular seu conteúdo, transformá-la em maiúsculas ou minúsculas, ou mesmo concatená-la com outras strings.

Criando Strings no Python

No Python, você pode criar strings de três maneiras principais:

1. Aspas Simples (' ')

Strings delimitadas por aspas simples são a forma mais básica de criar texto.

```
minha_string = 'Olá, mundo!'
```

2. Aspas Duplas (" ")

Funciona exatamente como aspas simples, mas é útil para incluir apóstrofes no texto sem erros de sintaxe.

```
outra_string = "Python é incrível, não é?"
```

3. Aspas Triplas (' ' ' ou " " ")

Permitem criar strings que ocupam várias linhas. Esse método é especialmente útil para escrever textos longos, como descrições ou mensagens multilineares.

```
string_multilinha = '''Python é uma linguagem de programação  
muito poderosa e versátil.  
Você pode aprender a usá-la para resolver muitos problemas!'''
```

Como próximo passo, você pode usar a função `print()` para exibir sua string na janela do console. Isso permite que você revise seu código e garanta que tudo funcione bem. Aqui está um trecho para isso:

```
print("Vamos imprimir uma string!")
```

Concatenação de Strings

Uma das primeiras habilidades que você pode dominar ao trabalhar com strings em Python é a **concatenação** — o processo de unir duas ou mais strings para formar uma única. Isso é feito usando o operador `+`. É simples, direto e uma ferramenta essencial para manipulação de texto.

Vamos começar com um exemplo básico:

```
string_um = "Olá, "  
string_dois = "mundo!"  
mensagem = string_um + string_dois  
print(mensagem)  # Saída: Olá, mundo!
```

Aqui, unimos duas strings (`string_um` e `string_dois`) para formar uma terceira string, `mensagem`. Esse processo é muito útil quando você precisa construir mensagens dinâmicas, como saudações personalizadas ou notificações para os usuários.

Dica: Evite Excesso de Concatenações

Embora a concatenação seja poderosa, usar muitas concatenações seguidas pode tornar o código menos eficiente e mais difícil de ler. Se você estiver combinando muitas partes de texto, considere outras abordagens, como **f-strings**, que veremos mais adiante.

Concatenando Strings com Números

Concatenar strings com números é um cenário comum, mas também pode ser uma armadilha para iniciantes. No Python, você **não pode** simplesmente somar uma string e um número. Isso resulta em um erro:

```
idade = 25
mensagem = "Minha idade é " + idade
# TypeError: can only concatenate str (not "int") to str
```

Por quê? Porque o Python trata strings e números como tipos de dados diferentes. Para combinar esses valores, é necessário converter o número em uma string usando a função `str()`.

Solução: Convertendo Números para Strings

Aqui está a maneira correta de concatenar strings com números:

```
idade = 25
mensagem = "Minha idade é " + str(idade)
print(mensagem) # Saída: Minha idade é 25
```

Esse exemplo usa a função `str()` para transformar o número `25` em uma string antes de concatená-lo com a mensagem. Vamos expandir esse conceito com mais exemplos:

```
# Exemplo 1: Concatenando com um número inteiro
quantidade = 3
item = "livros"
mensagem = "Eu comprei " + str(quantidade) + " " + item + "."
print(mensagem)  # Saída: Eu comprei 3 livros.
```

```
# Exemplo 2: Concatenando com um número de ponto flutuante
preco = 19.99
mensagem = "O preço final é R$ " + str(preco)
print(mensagem)  # Saída: O preço final é R$ 19.99
```

Erros Comuns

Um erro frequente entre iniciantes é esquecer de usar `str()` para converter números em strings, resultando no já conhecido `TypeError`. Para evitar isso:

- Sempre pergunte a si mesmo: "Estou tentando combinar texto com um número?"
- Quando a resposta for "sim", lembre-se de usar `str()` ou uma alternativa, como **f-strings**.

F-strings: Uma Maneira Melhor de Concatenar Strings

Introduzidas no Python 3.6, as **f-strings** (formatted string literals) oferecem uma maneira mais elegante e eficiente de criar strings formatadas. Elas eliminam a necessidade de usar `str()` explicitamente e tornam o código mais legível.

Como Funciona?

Ao usar uma f-string, você coloca a letra **f** antes das aspas que delimitam a string e insere variáveis ou expressões dentro de `{}` diretamente na string.

Veja o exemplo básico:

```
nome = "João"
idade = 30
mensagem = f"Meu nome é {nome} e eu tenho {idade} anos."
print(mensagem)  # Saída: Meu nome é João e eu tenho 30 anos.
```

Benefícios das F-strings

1. **Conversão Automática:** Não é necessário usar `str()`. Qualquer tipo de dado, como números ou booleanos, é automaticamente convertido para string.

```
preco = 49.99
mensagem = f"O preço do produto é R$ {preco}."
print(mensagem)  # Saída: O preço do produto é R$ 49.99.
```

2. **Legibilidade:** Variáveis e expressões são diretamente inseridas na string, tornando o código mais fácil de entender.

```
a = 5
b = 10
resultado = f"A soma de {a} e {b} é {a + b}."
print(resultado)  # Saída: A soma de 5 e 10 é 15.
```

3. **Desempenho:** F-strings são mais rápidas em comparação com concatenação usando `+` ou métodos como `str.format()`.
-

Exemplos Práticos de F-strings

Concatenando Strings com Números

```
quantidade = 7
produto = "chocolates"
preco = 3.50

mensagem = f"Eu comprei {quantidade} {produto}, cada um por R$ {preco}."
print(mensagem)  # Saída: Eu comprei 7 chocolates, cada um por R$ 3.5.
```

Expressões em F-strings

F-strings permitem incluir cálculos diretamente dentro das chaves:


```
desconto = 0.1 # 10%
preco_original = 50
preco_final = preco_original * (1 - desconto)

mensagem = f"O preço original era R$ {preco_original}, mas com desconto de {desconto*100}% ficou R$ {preco_final:.2f}."
print(mensagem)

# Saída: O preço original era R$ 50, mas com desconto de 10.0% ficou R$ 45.00.
```

Dicas para Trabalhar com F-strings

1. Use f-strings sempre que precisar concatenar variáveis com strings. Elas tornam o código mais limpo e eficiente.
2. Para textos longos, utilize f-strings multilineares com aspas triplas:

```
nome = "Alice"
cidade = "São Paulo"

mensagem = f"""Olá, {nome}!
Espero que você esteja gostando de morar em {cidade}.
Nos vemos em breve!"""
print(mensagem)
```

Conclusão

Concatenar strings é uma habilidade essencial em Python, mas o método escolhido pode impactar a clareza e a eficiência do código. Enquanto a concatenação com `+` é válida, as f-strings oferecem uma abordagem mais moderna e prática, especialmente ao lidar com variáveis de diferentes tipos. Com a prática, você logo verá que elas se tornam indispensáveis no seu dia a dia como programador.

Conversão de Tipagem

Em Python, a conversão de tipagem (ou casting) é o processo de converter um valor de um tipo de dado para outro. Isso é frequentemente necessário ao trabalhar com entradas do usuário ou quando precisamos garantir que os dados estejam no formato correto para operações específicas. A seguir, veremos como utilizar as funções integradas `int()`, `float()`, `str()` e `bool()` para realizar essas conversões.

1. Conversão para Inteiro (`int()`)

A função `int()` é usada para converter um valor em um número inteiro. Isso pode ser útil quando você precisa trabalhar com números inteiros em operações matemáticas.

```
# Convertendo uma string para inteiro
numero_str = "42"
numero_int = int(numero_str)
print(numero_int)  # Saída: 42

# Convertendo um número de ponto flutuante para inteiro
numero_float = 3.14
numero_int = int(numero_float)
print(numero_int)  # Saída: 3
```

Observações:

- Ao converter uma string que não representa um número, como `int("abc")`, ocorrerá um erro de `ValueError`.
- A conversão de um número de ponto flutuante para inteiro remove a parte decimal, não arredondando o número.

2. Conversão para Ponto Flutuante (`float()`)

A função `float()` converte um valor para um número de ponto flutuante. Isso é útil quando você precisa de precisão decimal em seus cálculos.

```
# Convertendo uma string para float
numero_str = "3.14"
numero_float = float(numero_str)
print(numero_float)  # Saída: 3.14

# Convertendo um inteiro para float
numero_int = 10
numero_float = float(numero_int)
print(numero_float)  # Saída: 10.0
```

Observações:

- Se a string não puder ser convertida para um número (por exemplo, `float("abc")`), ocorrerá um erro de `ValueError`.

3. Conversão para String (`str()`)

A função `str()` é usada para converter um valor em uma string. Isso é útil quando você deseja exibir informações ou concatenar diferentes tipos de dados.

Exemplo de Uso:

```
# Convertendo um inteiro para string
numero_int = 42
numero_str = str(numero_int)
print(numero_str)  # Saída: "42"

# Convertendo um float para string
numero_float = 3.14
numero_str = str(numero_float)
print(numero_str)  # Saída: "3.14"
```

Observações:

- A conversão para string pode ser feita em qualquer tipo de dado.

4. Conversão para Booleano (**bool()**)

A função **bool()** converte um valor em um booleano (**True** ou **False**). Em Python, quase todos os valores são considerados **True**, exceto os seguintes:

- O número **0** (zero)
- O valor **None**
- A string vazia **""**
- A lista vazia **[]**
- O dicionário vazio **{}**
- O conjunto vazio **set()**

```
# Convertendo um número para booleano
print(bool(0))      # Saída: False
print(bool(42))     # Saída: True

# Convertendo uma string para booleano
print(bool(""))     # Saída: False
print(bool("Olá"))  # Saída: True

# Convertendo uma lista para booleano
print(bool([]))     # Saída: False
print(bool([1, 2, 3])) # Saída: True
```

Observações:

- A conversão para booleano é útil em estruturas de controle e condições.

A conversão de tipagem é uma parte essencial do trabalho com dados em Python. As funções `int()`, `float()`, `str()` e `bool()` oferecem flexibilidade para manipular diferentes tipos de dados conforme necessário. É importante estar ciente das regras de conversão e possíveis erros que podem ocorrer ao tentar converter valores que não são compatíveis.

Agora que você conhece as funções de conversão de tipagem, pode usá-las para garantir que seus dados estejam no formato correto para qualquer operação que você deseja realizar em seu código!

Bora codar!

Chegando até aqui você já aprendeu muita coisa, então bora por tudo em prática com alguns exercícios para fixar bem o conteúdo.

Exemplo resolvido

Você foi contratado para criar um programa que converte temperaturas de graus Fahrenheit para graus Celsius. Para fazer isso, utilize a seguinte fórmula:

$$C = \frac{(F - 32) \times 5}{9}$$

Passo a passo da resolução:

1. **Entrada:** Receber a temperatura em graus Fahrenheit (F).
2. **Cálculo:** Aplicar a fórmula acima para encontrar a temperatura em graus Celsius (C).
3. **Saída:** Exibir a temperatura convertida.

Exemplo de Código:

```
# Recebe a temperatura em Fahrenheit e converte em float
fahrenheit = float(input("Digite a temperatura em graus Fahrenheit:
"))

# Calcula a temperatura em Celsius
celsius = (fahrenheit - 32) * 5 / 9

# Exibe o resultado
print(f"A temperatura em graus Celsius é: {celsius} °C")
```

Resolução:

Se o usuário digitar 68 graus Fahrenheit:

- Saída: A temperatura em graus Celsius é: 20.00 °C
-

Agora é sua vez 😊

1. Converter Temperatura em Graus Celsius para Fahrenheit

Maria é uma estudante de meteorologia que está fazendo uma pesquisa sobre o clima em diferentes países. Recentemente, ela recebeu dados de temperaturas em graus Celsius de várias regiões e precisa converter essas temperaturas para graus Fahrenheit para sua apresentação. Maria sabe que a fórmula para a conversão é:

$$F = C \times \frac{9}{5} + 32$$

Ajude Maria em sua pesquisa criando um programa em Python para facilitar esse trabalho, permitindo que ela insira a temperatura em graus Celsius e veja instantaneamente o valor correspondente em Fahrenheit.

2. Cálculo de Multa de Pesca

João Papo-de-Pescador é um amante da pesca e costuma passar os finais de semana na beira do lago, buscando capturar os maiores peixes possíveis. No entanto, ele esqueceu que, de acordo com a regulamentação do estado de São Paulo, a pesca está limitada a 50 quilos por dia. Se ele ultrapassar esse limite, terá que pagar uma multa de R\$ 4,00 por quilo excedente. Após um dia de pesca bem-sucedido, João precisa calcular quanto ele pescou e se vai precisar pagar alguma multa.

Faça um programa que o ajude a calcular a quantidade de quilos além do limite e o valor total da multa que ele deverá pagar.

3. Calcular Tempo de Download de um Arquivo

Ana é uma estudante que está preparando um projeto de pesquisa para a faculdade e precisa baixar um arquivo grande da internet. Ela está curiosa para

saber quanto tempo levará para concluir o download, já que a conexão de internet dela varia em velocidade. Sabendo que a fórmula para calcular o tempo de download em minutos é:

$$\text{Tempo (min)} = \frac{\text{Tamanho do arquivo (MB)}}{\text{Velocidade (Mbps)}} \times 8 \div 60$$

Criar um programa que aceita o tamanho do arquivo em megabytes e a velocidade de download em megabits por segundo para informar a ela o tempo aproximado que levará para baixar o arquivo.

4. Cálculo da Média de Notas

Lucas é um estudante do ensino médio e está nervoso com as notas finais da sua disciplina de matemática. Para saber se ele conseguiu passar na matéria, Lucas decide calcular a média das três últimas provas. Ele sabe que a média é obtida pela soma das notas dividida pelo número total de provas:

$$\text{Média} = \frac{\text{Nota 1} + \text{Nota 2} + \text{Nota 3}}{3}$$

Com isso, faça um programa que aceita as notas das três provas como entrada e calcula a média, para que ele possa verificar se obteve um desempenho satisfatório.

5. Conversão de Moeda

Fernanda está planejando uma viagem internacional e, como parte dos preparativos, precisa converter o orçamento que tem em Reais (R\$) para Dólares (US\$). Para isso, ela precisa da cotação do Dólar no momento da conversão. A fórmula que ela irá usar para fazer esse cálculo é:

$$\text{Valor em Dólares} = \frac{\text{Valor em Reais}}{\text{Cotação do Dólar}}$$

Escreva um programa que solicita o valor em Reais e a cotação do Dólar, e então calcula e exibe o valor equivalente em Dólares, facilitando assim o planejamento financeiro da viagem de Fernanda.

Instruções If-Else (Instruções Condicionais)

Imagine que você está dirigindo um carro. Quando você se aproxima de um semáforo, toma decisões com base na cor da luz: se está verde, você avança; se está vermelho, você para. Da mesma forma, instruções condicionais no Python permitem que você controle o fluxo de execução do programa com base em condições lógicas. São como "tomadas de decisão" no código.

Em Python, as condições básicas que você pode usar são:

- **Igualdade** (`==`): Verifica se dois valores são iguais.
- **Diferença** (`!=`): Verifica se dois valores são diferentes.
- **Menor que** (`<`): Verifica se um valor é menor que outro.
- **Menor ou igual a** (`<=`): Verifica se um valor é menor ou igual a outro.
- **Maior que** (`>`): Verifica se um valor é maior que outro.
- **Maior ou igual a** (`>=`): Verifica se um valor é maior ou igual a outro.

Essas condições são utilizadas para tomar decisões no código, e são amplamente empregadas com **instruções if, elif (else if) e else**.

A Simplicidade da Instrução `if`

Uma instrução `if` permite executar um bloco de código apenas se a condição especificada for **verdadeira**. Pense nisso como perguntar: "Devo fazer isso agora?". Se a resposta for sim (True), o código é executado.

```
if 5 > 1:  
    print("5 é maior que 1!")
```

Saída:

```
5 é maior que 1!
```

Aqui, a condição `5 > 1` é avaliada como **True**, então o Python executa a instrução `print`.

Erros Comuns em Iniciantes

- **Esquecer a indentação:** O Python depende de indentação para determinar o que está dentro de um bloco condicional.

Exemplo errado:

```
if 5 > 1:  
print("Isso não funcionará!") # Indentação faltando
```

Resultado:

```
IndentationError: expected an indented block
```

- **Esquecer os dois pontos (:):** As condições precisam terminar com dois pontos.

Instruções `if` Aninhadas

Assim como decisões podem levar a outras decisões na vida real, no Python você pode aninhar instruções `if`. Isso significa que você pode verificar uma condição dentro de outra.

```
x = 35
if x > 20:
    print("Acima de vinte,")
    if x > 30:
        print("e também acima de 30!")
```

Saída:

```
Acima de vinte,
e também acima de 30!
```

Aqui, o primeiro `if` verifica se `x > 20`. Se for verdadeiro, ele entra no bloco interno e verifica se `x > 30`. Esse tipo de estrutura é útil quando há condições hierárquicas.

Dica:

Evite aninhar muitos `if` seguidos, pois isso pode tornar o código confuso. Em vez disso, use operadores lógicos, como `and` e `or`, para combinar condições.

Introdução ao `elif`

Às vezes, você precisa verificar várias condições diferentes. É aqui que entra a instrução `elif`. Ela é como perguntar: "E se essa outra condição for verdadeira?"


```
a = 45
b = 45
if b > a:
    print("b é maior que a")
elif a == b:
    print("a e b são iguais")
```

Saída:

```
a e b são iguais
```

No exemplo, o Python verifica as condições na ordem:

1. `b > a`: Não é verdadeiro.
2. `a == b`: Verdadeiro, então ele executa o bloco associado.

Dica:

- Você pode usar quantos `elif` precisar.
 - Apenas o **primeiro bloco verdadeiro** é executado, mesmo que outras condições também sejam verdadeiras.
-

Adicionando o `else`

E se nenhuma das condições `if` ou `elif` forem verdadeiras? É aqui que entra o `else`. Ele age como um "plano de contingência" para cobrir todos os outros cenários possíveis.

```
idade = 16
if idade < 4:
    preco_ingresso = 0
elif idade < 18:
    preco_ingresso = 10
else:
    preco_ingresso = 15

print(f"O preço do ingresso é R$ {preco_ingresso}.")
```

Saída:

```
O preço do ingresso é R$ 10.
```

Aqui, o Python avalia as condições na ordem:

1. `idade < 4`: Falso.
 2. `idade < 18`: Verdadeiro, então define `preco_ingresso` como 10 e pula o `else`.
-

Negando Condições com `not`

O operador `not` é útil quando você precisa verificar o oposto de uma condição. É como perguntar: "Isso não é verdade?".

```
nova_lista = [1, 2, 3, 4]
x = 10
if x not in nova_lista:
    print("'x' não está na lista, então isso é Verdadeiro!")
```

Saída:

```
'x' não está na lista, então isso é Verdadeiro!
```

Erros Comuns

- Esquecer que `not` só inverte o valor lógico. Não altera os elementos diretamente. Por exemplo:

```
print(not 0)    # True, porque 0 é considerado falso.
print(not 1)    # False, porque 1 é considerado verdadeiro.
```

Lidando com Condições Vazias: `pass`

Às vezes, você sabe que precisará de uma instrução `if`, mas ainda não definiu o que ela deve fazer. Nesse caso, você pode usar `pass` como um "marcador" para evitar erros de sintaxe.

```
a = 33
b = 200
if b > a:
    pass # Código ainda será implementado aqui
```

O código acima não faz nada, mas evita um erro de execução.

Boas Práticas ao Usar Condicionais

1. **Mantenha o Código Simples:** Tente evitar muitos níveis de `if` aninhados. Use operadores lógicos para combinar condições:

```
if idade > 18 and idade < 60:
    print("Você está na faixa etária ideal para trabalhar.")
```

2. **Use Nomes de Variáveis Claros:** Evite nomes genéricos como `a`, `b`. Eles dificultam a leitura do código:

```
if preco_produto < 100:
    print("Produto acessível!")
```

3. **Testar Todos os Cenários:** Certifique-se de cobrir todos os possíveis valores de entrada para evitar resultados inesperados.
4. **Evite Condições Desnecessárias:** Por exemplo, em vez de:

```
if x == True:  
    print("Verdadeiro!")
```

Apenas escreva:

```
if x:  
    print("Verdadeiro!")
```

Conclusão

As instruções condicionais são como "placas de sinalização" em um programa, orientando o fluxo do código com base em condições lógicas. Usando `if`, `elif`, `else`, e operadores como `not`, você pode criar programas que tomam decisões inteligentes e dinâmicas. Com prática, essas construções se tornarão naturais, permitindo que você crie códigos mais poderosos e eficientes.

Bora codar!

Agora que você já entendeu como usar condicionais no Python, vamos colocar em prática com exercícios para reforçar o aprendizado. Cada exercício apresenta um cenário simples e realista, onde você aplicará as instruções condicionais para resolver problemas.

1. Verificar Maioridade

Carlos é um porteiro de um edifício e precisa verificar a idade dos visitantes para garantir que todos sejam maiores de 18 anos antes de entrarem na área de lazer. Crie um programa que:

- Receba a idade de um visitante.
- Verifique se ele é maior de idade.
- Exiba uma mensagem informando se a entrada foi permitida ou não.

Dica: Use uma instrução `if-else`.

2. Calcular Aumento Salarial

Paula é uma gerente de uma empresa e precisa implementar uma regra de aumento salarial. A regra é a seguinte:

- Se o salário atual for menor que R\$ 2.000,00, o aumento será de 15%.
- Caso contrário, o aumento será de 10%.

Crie um programa que:

- Receba o salário atual de um funcionário.
- Calcule o novo salário com base nas regras acima.
- Exiba o novo salário.

Dica: Use uma instrução `if-else`.

3. Determinar o Estado da Água

Você é um pesquisador de química e precisa desenvolver um programa para determinar o estado físico da água com base na temperatura fornecida. Considere:

- Temperatura abaixo de 0°C: Água no estado sólido.
- Temperatura entre 0°C e 100°C (inclusive): Água no estado líquido.
- Temperatura acima de 100°C: Água no estado gasoso.

Tarefa:

- Crie um programa que receba a temperatura e exiba o estado físico correspondente.

Dica: Use `if`, `elif` e `else`.

4. Calculadora de IMC

João quer criar um programa para calcular seu Índice de Massa Corporal (IMC). A fórmula para calcular o IMC é:

$$IMC = \frac{\text{peso (kg)}}{\text{altura (m)}^2}$$

E as classificações são:

- Abaixo de 18.5: Abaixo do peso.
- Entre 18.5 e 24.9: Peso normal.
- Entre 25 e 29.9: Sobrepeso.
- 30 ou mais: Obesidade.

Tarefa:

- Crie um programa que receba o peso e a altura de uma pessoa, calcule o IMC e exiba a classificação correspondente.

Dica: Use `if`, `elif` e `else` para as classificações.

5. Cálculo de Passagem de Ônibus

Ana quer viajar para outra cidade e precisa calcular o custo da passagem de ônibus. As regras são:

- Até 200 km: R\$ 0,50 por quilômetro.
- Acima de 200 km: R\$ 0,45 por quilômetro.

Tarefa:

- Crie um programa que receba a distância da viagem em quilômetros.
- Calcule o custo da passagem com base nas regras acima.
- Exiba o valor total da passagem.

Dica: Use `if-else` para decidir a tarifa aplicável.

6. Par ou Ímpar

Lucas está aprendendo sobre números pares e ímpares na escola e quer um programa que o ajude a verificar isso rapidamente. Crie um programa que:

- Solicite um número inteiro ao usuário.
- Informe se o número é par ou ímpar.

Dica: Use o operador `%` para verificar o resto da divisão por 2.

7. Desconto em Compras

Uma loja está oferecendo descontos baseados no valor total da compra:

- Compras acima de R\$ 500,00 têm 10% de desconto.
- Compras entre R\$ 200,00 e R\$ 500,00 (inclusive) têm 5% de desconto.
- Compras abaixo de R\$ 200,00 não têm desconto.

Tarefa:

- Crie um programa que receba o valor total da compra.
- Calcule e exiba o valor do desconto e o valor final após o desconto.

Dica: Use `if`, `elif` e `else`.

8. Calculadora de Multa de Velocidade

Um policial rodoviário quer um programa que calcule a multa para motoristas que ultrapassam o limite de velocidade. As regras são:

- Velocidade até 80 km/h: Sem multa.
- Acima de 80 km/h: Multa de R\$ 5,00 por km excedido.

Tarefa:

- Receba a velocidade do carro.
- Calcule e exiba o valor da multa (se aplicável).

Dica: Subtraia a velocidade máxima permitida para determinar o excesso.

9. Verificar um Número Primo

Crie um programa que receba um número inteiro e determine se ele é primo. Um número primo é aquele que só pode ser dividido por 1 e por ele mesmo.

Tarefa:

- Solicite um número do usuário.
- Verifique se ele é primo.
- Exiba uma mensagem informando o resultado.

Dica: Use um laço para testar divisores entre 2 e o número.

10. Calculadora de Ano Bissexto

Um ano é bissexto se:

- For divisível por 4, mas não por 100, **ou**
- For divisível por 400.

Tarefa:

- Crie um programa que receba um ano como entrada.
- Verifique se ele é bissexto.
- Exiba uma mensagem informando o resultado.

Dica: Combine operadores lógicos para testar as condições.

Loops em Python

Loops são uma das ferramentas mais poderosas em programação, permitindo que você repita um bloco de código várias vezes, economizando tempo e reduzindo a complexidade do programa. Em Python, os dois tipos principais de loops são:

1. **Loops `for`** - usados para iterar sobre sequências como listas, strings e tuplas.
2. **Loops `while`** - usados quando você deseja repetir um bloco de código enquanto uma condição é verdadeira.

Vamos explorar cada um deles em detalhes.

Loop For: Iterando Sobre Sequências

O loop `for` é ideal para percorrer elementos de uma sequência (como listas, strings, dicionários ou tuplas). Ele funciona como um iterador, pegando cada elemento da sequência, um de cada vez, e executando o bloco de código correspondente.

Exemplo Simples

Vamos começar com um exemplo básico usando uma string:

```
for letra in "python":  
    print(letra)
```

Saída:

```
p  
y  
t  
h  
o  
n
```

Aqui, a variável `letra` recebe cada caractere da string `"python"` em cada iteração, e o `print` exibe esse caractere.

Iterando Sobre Listas

Os loops `for` são muito úteis para trabalhar com listas. Veja este exemplo:

```
frutas = ["maçã", "banana", "cereja"]  
for fruta in frutas:  
    print(fruta)
```

Saída:

```
maçã  
banana  
cereja
```

Cada elemento da lista `frutas` é atribuído à variável `fruta` em cada iteração.

Usando `range()` em Loops For

O Python fornece a função `range()` para gerar sequências de números, o que é muito útil em loops.

```
for numero in range(5):  
    print(numero)
```

Saída:

```
0  
1  
2  
3  
4
```

Por padrão, `range(5)` gera números de 0 a 4. Você também pode especificar um intervalo personalizado:

```
for numero in range(2, 8):  
    print(numero)
```

Saída:

```
2  
3  
4  
5  
6  
7
```

Você pode até especificar um "passo" para pular números:

```
for numero in range(0, 10, 2):  
    print(numero)
```

Saída:

```
0  
2  
4  
6  
8
```

Iterando Sobre Dicionários

Com dicionários, você pode iterar pelas **chaves**, **valores** ou ambos:

```
dados = {"nome": "Ana", "idade": 25, "cidade": "São Paulo"}
```

```
# Iterar pelas chaves
```

```
for chave in dados:  
    print(chave)
```

```
# Iterar pelos valores
```

```
for valor in dados.values():  
    print(valor)
```

```
# Iterar por chave e valor
```

```
for chave, valor in dados.items():  
    print(f"{chave}: {valor}")
```

Loop While: Executando Enquanto a Condição For Verdadeira

O loop **while** repete um bloco de código enquanto uma condição é avaliada como verdadeira. Ele é mais adequado para cenários onde você não sabe exatamente quantas vezes o loop será executado.

Exemplo Básico

```
contador = 1
while contador ≤ 5:
    print(f"Contando: {contador}")
    contador += 1
```

Saída:

```
Contando: 1
Contando: 2
Contando: 3
Contando: 4
Contando: 5
```

Aqui, o loop continua enquanto `contador` for menor ou igual a 5. A variável `contador` é incrementada em cada iteração.

Evitando Loops Infinitos

Um erro comum para iniciantes é criar loops infinitos, onde a condição nunca se torna falsa. Isso pode travar o programa. Por exemplo:

```
contador = 1
while contador > 0:
    print("Este é um loop infinito!")
```

Aqui, a condição `contador > 0` nunca será falsa, resultando em um loop infinito. Para

evitar isso, certifique-se de que a variável de controle (**contador**, no caso) seja alterada dentro do loop.

Como Quebrar um Loop: Usando **break**

A instrução **break** permite que você saia de um loop antes que ele complete todas as iterações.

Exemplo com Loop While

```
i = 1
while i <= 10:
    print(i)
    if i == 5:
        break
    i += 1
```

Saída:

```
1
2
3
4
5
```

Quando **i** chega a 5, o loop é interrompido.

Exemplo com Loop For

```
for letra in "python":  
    if letra == "h":  
        break  
    print(letra)
```

Saída:

```
p  
y  
t
```

O loop é interrompido assim que encontra o caractere "h".

Pulando Iterações: Usando `continue`

A instrução `continue` faz com que o loop pule a iteração atual e passe para a próxima.

Exemplo com Loop While

```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue
    print(i)
```

Saída:

```
1
3
5
7
9
```

Aqui, números pares são ignorados por causa do `continue`.

Exemplo com Loop For

```
for numero in range(10):
    if numero % 2 == 0:
        continue
    print(numero)
```

Saída:

1
3
5
7
9

Exemplos Práticos

1. Soma de Números de 1 a 10

```
soma = 0
for numero in range(1, 11):
    soma += numero
print(f"A soma dos números de 1 a 10 é: {soma}")
```

2. Encontrar Palavras em um Texto

```
texto = "Python é incrível"
for palavra in texto.split():
    if palavra == "incrível":
        print("Encontrei a palavra 'incrível!'")
```

3. Jogo de Adivinhação


```
numero_secreto = 7
while True:
    tentativa = int(input("Adivinhe o número (1 a 10): "))
    if tentativa == numero_secreto:
        print("Parabéns! Você acertou!")
        break
    else:
        print("Tente novamente.")
```

Conclusão

Loops são uma ferramenta indispensável para tarefas repetitivas e manipulação de sequências. O loop **for** é ideal para iterar sobre coleções, enquanto o loop **while** é perfeito para condições indefinidas. Com prática, você dominará essas estruturas e será capaz de escrever códigos mais eficientes e dinâmicos!

Definindo Funções em Python

Funções são blocos de código que realizam tarefas específicas. Elas permitem reutilizar código, tornando os programas mais organizados, legíveis e fáceis de manter. No Python, você pode definir suas próprias funções para encapsular lógica e evitar repetição.

Vamos explorar, em detalhes, como criar, chamar e usar funções, além de entender conceitos como parâmetros e argumentos.

O Que é uma Função?

Uma função é uma espécie de "máquina" em seu programa. Ela:

1. Recebe **entradas** (opcional).
2. Processa essas entradas.
3. Retorna uma **saída** (opcional).

Por exemplo, imagine uma função como um caixa eletrônico:

- Você insere seu cartão e digita um valor (entrada).
 - O sistema processa o pedido.
 - Você recebe dinheiro ou informações da conta (saída).
-

Como Definir uma Função

Para criar uma função no Python:

1. Use a palavra-chave **def**.

2. Dê um **nome descritivo** para a função.
3. Adicione **parênteses** `()` — com ou sem parâmetros dentro.
4. Termine a linha com dois pontos `:`.
5. Indente o bloco de código que representa o corpo da função (4 espaços por padrão).

Exemplo Básico: Definir e Chamar uma Função

```
def saudacao()  
    print("Olá! Seja bem-vindo!")
```

Aqui, definimos uma função chamada `saudacao`. Para usá-la, precisamos chamá-la:

```
saudacao()
```

Saída:

```
Olá! Seja bem-vindo!
```

Componentes de uma Função

Nome da Função

Escolha nomes significativos que expliquem o propósito da função. Por exemplo, em vez de `def func1():`, prefira algo como `def calcular_media():`.

Parâmetros e Argumentos

- **Parâmetros:** São as variáveis que você define na função, dentro dos parênteses. Eles atuam como "canais" para passar informações para a função.
- **Argumentos:** São os valores reais fornecidos quando você chama a função.

Corpo da Função

O corpo da função contém as instruções que ela executará. Ele deve ser indentado para indicar que pertence à função.

Funções com Parâmetros

Uma função pode receber parâmetros para trabalhar com diferentes valores. Veja este exemplo:

```
def saudacao_personalizada(nome)
    print(f"Olá, {nome}! Seja bem-vindo!")
```

Quando chamada, você fornece um argumento correspondente ao parâmetro `nome`:

```
saudacao_personalizada("Maria")
```

Saída:

```
Olá, Maria! Seja bem-vindo!
```

Você também pode passar diferentes valores para o mesmo código:

```
saudacao_personalizada("João")  
saudacao_personalizada("Ana")
```

Saída:

```
Olá, João! Seja bem-vindo!  
Olá, Ana! Seja bem-vindo!
```

Funções com Múltiplos Parâmetros

Funções podem ter vários parâmetros. Vamos criar uma função que soma três números:

```
def somar_numeros(x, y, z):  
    resultado = x + y + z  
    print(f"A soma de {x}, {y} e {z} é {resultado}.")
```

Agora, podemos chamar a função e passar três argumentos:

```
somar_numeros(1, 2, 3)
```

Saída:

```
A soma de 1, 2 e 3 é 6.
```

Dica:

Os parâmetros devem ser passados na **ordem correta**, a menos que sejam usados argumentos nomeados (que veremos mais adiante).

Funções com Retorno

Nem todas as funções precisam exibir resultados imediatamente. Algumas podem simplesmente retornar um valor para que você o use em outro lugar do programa. Para isso, usamos a palavra-chave **return**.

```
def calcular_area_retangulo(largura, altura):  
    return largura * altura
```

Aqui, a função retorna o resultado da multiplicação, mas não o imprime. Para usar esse valor, fazemos o seguinte:

```
area = calcular_area_retangulo(5, 10)
print(f"A área do retângulo é {area}.")
```

Saída:

```
A área do retângulo é 50.
```

Argumentos Nomeados

Argumentos nomeados permitem que você especifique os valores de entrada em qualquer ordem, associando-os explicitamente aos parâmetros. Isso é útil para melhorar a legibilidade e evitar erros.

Exemplo:

```
def informacoes_produto(nome_produto, preco):
    print(f"Produto: {nome_produto}")
    print(f"Preço: R$ {preco:.2f}")
```

Chamando a função com argumentos nomeados:

```
informacoes_produto(nome_produto="Camisa", preco=39.90)
informacoes_produto(preco=59.99, nome_produto="Calça")
```

Saída:

```
Produto: Camisa  
Preço: R$ 39.90  
Produto: Calça  
Preço: R$ 59.99
```

Vantagem:

A ordem dos argumentos não importa ao usar nomes explícitos.

Exemplos Práticos

Exemplo 1: Calculadora de Desconto

```
def calcular_preco_com_desconto(preco, desconto):  
    preco_final = preco - (preco * (desconto / 100))  
    return preco_final  
  
preco_inicial = 100  
desconto = 20  
preco_final = calcular_preco_com_desconto(preco_inicial,  
desconto)  
print(f"O preço com {desconto}% de desconto é R$ {preco_final:.2f}.")
```

Saída:

O preço com 20% de desconto é R\$ 80.00.

Exemplo 2: Verificar Idade

```
def verificar_maioridade(idade):  
    if idade ≥ 18:  
        return "Maior de idade"  
    else:  
        return "Menor de idade"  
  
print(verificar_maioridade(20)) # Saída: Maior de idade  
print(verificar_maioridade(16)) # Saída: Menor de idade
```

Boas Práticas ao Criar Funções

1. **Nomes Significativos:** Escolha nomes que descrevam claramente o que a função faz.
 - Exemplo ruim: `def func1():`
 - Exemplo bom: `def calcular_media():`
2. **Comentários:** Explique o que a função faz, especialmente se for complexa.

```
def calcular_media(nota1, nota2)
    """Calcula a média de duas notas."""
    return (nota1 + nota2) / 2
```

3. **Evite Funções Muito Longas:** Se uma função está ficando muito grande, considere dividi-la em funções menores.
 4. **Teste Suas Funções:** Teste a função com diferentes entradas para garantir que funcione em todos os casos.
-

Conclusão

Funções são essenciais para criar programas reutilizáveis e organizados. Com elas, você pode encapsular lógica complexa e usá-la repetidamente, economizando tempo e esforço. O próximo passo é praticar: tente criar suas próprias funções para resolver problemas do mundo real e experimente passar parâmetros, usar retornos e explorar argumentos nomeados. Com o tempo, as funções se tornarão suas ferramentas favoritas na programação!

Coleções de Dados

Listas

Listas são outro tipo de dado fundamental em Python usado para especificar uma sequência ordenada de elementos. Em resumo, elas ajudam você a manter dados relacionados juntos e realizar as mesmas operações em vários valores ao mesmo tempo. Ao contrário das strings, as listas são mutáveis (= alteráveis).

Cada valor dentro de uma lista é chamado de item e esses itens são colocados entre colchetes.

Exemplos de Listas

```
minha_lista = [1, 2, 3]
minha_lista2 = ["a", "b", "c"]
minha_lista3 = ["4", "d", "livro", 5]
```

Alternativamente, você pode usar a função `list()` para fazer o mesmo:

```
lista_alpha = list(("1", "2", "3"))
print(lista_alpha)
```

Como Adicionar Itens a uma Lista

Você tem duas maneiras de adicionar novos itens a listas existentes.

A primeira é usando a função `append()`:

```
lista_beta = ["maçã", "banana", "laranja"]
lista_beta.append("uva")
print(lista_beta)
```

A segunda opção é usar a função `insert()` para adicionar um item no índice especificado:

```
lista_beta = ["maçã", "banana", "laranja"]
lista_beta.insert(2, "uva")
print(lista_beta)
```

Como Remover um Item de uma Lista

Novamente, você tem várias maneiras de fazer isso. Primeiro, você pode usar a função `remove()`:

```
lista_beta = ["maçã", "banana", "laranja"]
lista_beta.remove("maçã")
print(lista_beta)
```

Em segundo lugar, você pode usar a função `pop()`. Se nenhum índice for especificado, ele removerá o último item.

```
lista_beta = ["maçã", "banana", "laranja"]
lista_beta.pop()
print(lista_beta)
```

A última opção é usar a palavra-chave `del` para remover um item específico:

```
lista_beta = ["maçã", "banana", "laranja"]  
del lista_beta[1]  
print(lista_beta)
```

P.S. Você também pode aplicar `del` para toda a lista para eliminá-la.

Combinar Duas Listas

Para mesclar duas listas, use o operador `+`.

```
minha_lista = [1, 2, 3]  
minha_lista2 = ["a", "b", "c"]  
lista_comb = minha_lista + minha_lista2  
print(lista_comb)  # Saída: [1, 2, 3, 'a', 'b', 'c']
```

Criar uma Lista Aninhada

Você também pode criar uma lista de suas listas quando tiver muitas delas



```
minha_lista_aninhada = [minha_lista, minha_lista2]
print(minha_lista_aninhada)  # Saída: [[1, 2, 3], ['a', 'b', 'c']]
```

Ordenar uma Lista

Use a função `sort()` para organizar todos os itens na sua lista.

```
lista_alpha = [34, 23, 67, 100, 88, 2]
lista_alpha.sort()
print(lista_alpha)  # Saída: [2, 23, 34, 67, 88, 100]
```

Fatiar uma Lista

Agora, se você quiser chamar apenas alguns elementos da sua lista (por exemplo, os primeiros 4 itens), precisa especificar um intervalo de números de índice separados por dois pontos `[x:y]`. Aqui está um exemplo:

```
print(lista_alpha[0:4])  # Saída: [2, 23, 34, 67]
```

Alterar o Valor de um Item na Sua Lista

Você pode facilmente sobrescrever um valor de um item de lista:

```
lista_beta = ["maçã", "banana", "laranja"]  
lista_beta[1] = "pera"  
print(lista_beta)  # Saída: ['maçã', 'pera', 'laranja']
```

Looping Através da Lista

Usando o loop `for`, você pode multiplicar o uso de certos itens, semelhante ao que o operador `*` faz. Aqui está um exemplo:

```
for x in range(1, 4):  
    lista_beta += ['fruta']  
print(lista_beta)
```

Copiar uma Lista

Use a função embutida `copy()` para replicar seus dados:

```
lista_beta = ["maçã", "banana", "laranja"]  
lista_beta = lista_beta.copy()  
print(lista_beta)
```

Alternativamente, você pode copiar uma lista com o método `list()`:

```
lista_beta = ["maçã", "banana", "laranja"]  
lista_beta = list(lista_beta)  
print(lista_beta)
```

Compreensões de Listas

As compreensões de listas são uma opção útil para criar listas com base em listas existentes. Ao usá-las, você pode construir usando strings e tuplas também.

Exemplos de Compreensões de Listas

```
variavel_lista = [x for x in iteravel]
```

Aqui está um exemplo mais complexo que apresenta operadores matemáticos, inteiros e a função `range()`:

```
lista_numeros = [x ** 2 for x in range(10) if x % 2 == 0]  
print(lista_numeros)
```


Tuplas

As tuplas são semelhantes às listas — elas permitem que você exiba uma sequência ordenada de elementos. No entanto, elas são imutáveis e você não pode mudar os valores armazenados em uma tupla.

A vantagem de usar tuplas em vez de listas é que as primeiras são um pouco mais rápidas. Portanto, é uma boa maneira de otimizar seu código.

Como Criar uma Tupla

```
minha_tupla = (1, 2, 3, 4, 5)
print(minha_tupla[0:3]) # Saída: (1, 2, 3)
```

Nota: Uma vez que você cria uma tupla, não pode adicionar novos itens a ela ou alterá-la de nenhuma outra maneira!

Como Fatiar uma Tupla

O processo é semelhante ao de fatiar listas.

```
numeros = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
print(numeros[1:11:2]) # Saída: (1, 3, 5, 7, 9)
```

Converter Tupla em Lista

Como as tuplas são imutáveis, você não pode alterá-las. O que você pode fazer, no entanto, é converter uma tupla em uma lista, fazer uma edição e depois convertê-la de volta em uma tupla. Aqui está como fazer isso:

```
python
```

```
x = ("maçã", "laranja", "pera")
```

```
y = list(x)
```

```
y[1] = "uva"
```

```
x = tuple(y)
```

```
print(x) # Saída: ('maçã', 'uva', 'pera')
```

Dicionários

Um dicionário contém índices com chaves que são mapeadas para certos valores. Esses pares chave-valor oferecem uma ótima maneira de organizar e armazenar dados em Python. Eles são mutáveis, o que significa que você pode alterar as informações armazenadas.

Um valor chave pode ser uma string, Booleano ou inteiro. Aqui está um exemplo de dicionário ilustrando isso:

```
cliente1 = {'usuario': 'john-sea', 'online': False, 'amigos': 100}
```

Como Criar um Dicionário Python

Aqui está um exemplo rápido mostrando como fazer um dicionário vazio. Opção 1:

```
novo_dicionario = {}
```

Opção 2:

```
outro_dicionario = dict()
```

E você pode usar as mesmas duas abordagens para adicionar valores ao seu dicionário:

```
novo_dicionario = {  
    "marca": "Honda",  
    "modelo": "Civic",  
    "ano": 1995  
}  
print(novo_dicionario)
```

Como Acessar um Valor em um Dicionário

Você pode acessar qualquer um dos valores em seu dicionário da seguinte maneira:

```
x = novo_dicionario["marca"]
```

Você também pode usar os seguintes métodos para alcançar o mesmo objetivo.

- `dict.keys()` isola as chaves
- `dict.values()` isola os valores
- `dict.items()` retorna itens em um formato de lista de pares (chave, valor)

Alterar Valor de Item

Para alterar um dos itens, você precisa se referir a ele pelo nome da chave:

```
# Alterar o "ano" para 2020:
novo_dicionario = {
    "marca": "Honda",
    "modelo": "Civic",
    "ano": 1995
}
novo_dicionario["ano"] = 2020
```

Looping Através do Dicionário

Mais uma vez, para implementar loops, use o comando de loop **for**. **Nota:** Neste caso, os valores de retorno são as chaves do dicionário. Mas, você também pode retornar valores usando outro método.

```
# Imprima todos os nomes das chaves no dicionário
for x in novo_dicionario:
    print(x)

# Imprima todos os valores no dicionário
for x in novo_dicionario:
    print(novo_dicionario[x])

# Loop através de chaves e valores
for x, y in novo_dicionario.items():
    print(x, y)
```

Tratando Exceções (Erros) em Python

O Python possui uma lista de exceções embutidas (erros) que aparecerão sempre que você cometer um erro no seu código. Como novato, é bom saber como corrigir esses erros.

As Exceções Python Mais Comuns

- **AttributeError** — aparece quando uma referência ou atribuição de atributo falha.
- **IOError** — surge quando alguma operação de I/O (por exemplo, a função `open()`) falha por um motivo relacionado a I/O, como "arquivo não encontrado" ou "disco cheio".
- **ImportError** — ocorre quando uma instrução de importação não pode localizar a definição do módulo. Também quando um `from ... import` não pode encontrar um nome que deve ser importado.
- **IndexError** — surge quando um subscrito de sequência está fora do intervalo.
- **KeyError** — levantada quando uma chave de

dicionário não é encontrada no conjunto de chaves existentes.

- **KeyboardInterrupt** — acende quando o usuário pressiona a tecla de interrupção (como Control-C ou Delete).
- **NameError** — aparece quando um nome local ou global não pode ser encontrado.
- **OSError** — indica um erro relacionado ao sistema.
- **SyntaxError** — aparece quando um parser encontra um erro de sintaxe.
- **TypeError** — surge quando uma operação ou função é aplicada a um objeto de tipo inadequado.
- **ValueError** — levantada quando uma operação/função embutida recebe um argumento que tem o tipo certo, mas não um valor apropriado.
- **ZeroDivisionError** — surge quando o segundo argumento de uma operação de divisão ou módulo é zero.

Como Solucionar Erros

O Python tem uma declaração útil, projetada especificamente para o propósito de lidar

com exceções — a declaração `try/except`. Aqui está um trecho de código mostrando como você pode capturar `KeyErrors` em um dicionário usando essa declaração:

```
meu_dict = {"a": 1, "b": 2, "c": 3}
try:
    valor = meu_dict["d"]
except KeyError:
    print("Essa chave não existe!")
```

Você também pode detectar várias exceções ao mesmo tempo com uma única declaração. Aqui está um exemplo para isso:

```
meu_dict = {"a": 1, "b": 2, "c": 3}
try:
    valor = meu_dict["d"]
except IndexError:
    print("Esse índice não existe!")
except KeyError:
    print("Essa chave não está no dicionário!")
except:
    print("Outro problema aconteceu!")
```

`try/except` com cláusula `else`

Adicionar uma cláusula `else` ajudará você a confirmar que nenhum erro foi encontrado:

```
meu_dict = {"a": 1, "b": 2, "c": 3}

try:
    valor = meu_dict["a"]
except KeyError:
    print("Ocorreu um KeyError!")
else:
    print("Nenhum erro ocorreu!")
```

Conclusões

Agora você conhece os conceitos principais do Python!

De forma alguma esta lista de verificação do Python é abrangente. Mas inclui todos os principais tipos de dados, funções e comandos que você deve aprender como iniciante.

