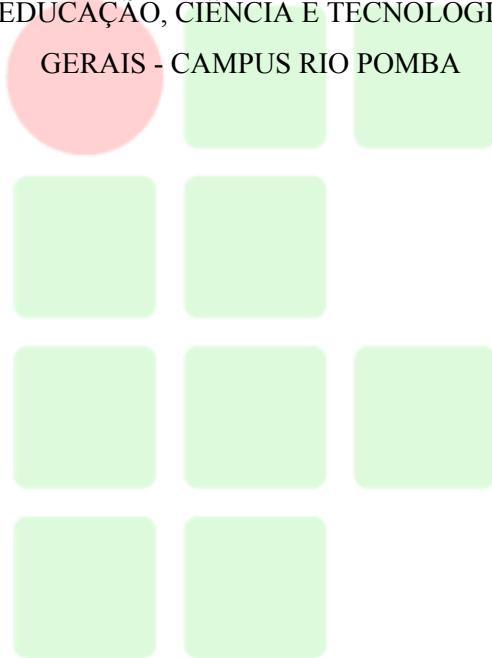


INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO SUDESTE DE MINAS
GERAIS - CAMPUS RIO POMBA



APOSTILA: INTRODUÇÃO AO PYTHON

**INSTITUTO
FEDERAL**

Sudeste de
Minas Gerais

Campus
Rio Pomba

Rio Pomba - 2025

SUMÁRIO

Agradecimentos.....	5
CAPÍTULO 1: Introdução.....	6
1.1 História do Python.....	6
1.2 Paradigmas.....	7
1.3 Classificação da linguagem quanto a ortografia.....	8
1.4 Classificação da linguagem quanto a tipagem.....	8
1.5 Classificação da linguagem quanto a execução.....	9
1.6 Exercícios.....	9
CAPÍTULO 2: Ambiente de Desenvolvimento Windows.....	10
2.1 Objetivos.....	10
2.2 Instalação do Python.....	10
2.3 Instalação do VS Code.....	11
2.4 Configurações do VS Code.....	12
CAPÍTULO 3: Variáveis e Tipos de Dados.....	12
3.1 Variáveis.....	12
3.2 Tipos de dados.....	12
3.3 Padrões de nomenclatura.....	13
3.4 Declarando variáveis.....	13
3.5 Dicas importantes.....	13
3.6 Exercícios.....	14
CAPÍTULO 4: Operações Aritméticas e Lógicas.....	15
4.1 Operações Aritméticas.....	15
4.2 Operações Lógicas.....	16
4.3 Operadores de Atribuição.....	19
4.4 Exercícios.....	21
CAPÍTULO 5: Estruturas de Controle de Fluxo 'if', 'elif' e 'else'.....	22
5.1 Importância da Indentação.....	23
5.2 Exercícios.....	23
CAPÍTULO 6: Estruturas de Controle de Fluxo 'for' e 'while'.....	24
6.1 Loop 'for' em Python.....	25
6.2 Loop 'while' em Python.....	25
6.3 Exercícios.....	26
CAPÍTULO 7: Funções.....	27
7.1 Funções úteis.....	28
7.2 Exercícios.....	29
CAPÍTULO 8: Listas, Tuplas e Dicionários.....	30
8.1 Listas.....	30
8.1.1 Adicionando elementos nas listas.....	31
8.1.2 Removendo elementos das listas.....	32
8.1.3 Modificando elementos de uma lista.....	32
8.1.4 Somando todos os números de uma lista.....	32

8.1.5 Descobrimos o tamanho de uma lista.....	33
8.1.6 Descobrimos o número de vezes que um elemento aparece na lista.....	33
8.1.7 Utilizando o loop for em listas.....	33
8.1.8 Manipulando elementos com o loop for.....	33
8.1.9 Usando a função enumerate() no loop for.....	34
8.1.10 Usando while em listas.....	34
8.1.11 Exemplos Práticos de Loops com Listas.....	35
8.2 Tuplas.....	36
8.2.1 Por quê usar tuplas?.....	36
8.2.1 Operações com Tuplas.....	37
8.2.2 Desempacotamento de Tuplas.....	38
8.3 Dicionários.....	39
8.3.1 Por quê usar dicionários?.....	39
8.3.2 Criando dicionários.....	39
8.3.3 Acessando valores.....	40
8.3.4 Modificando dicionários.....	40
8.3.4 Iterando sobre dicionários.....	41
8.3.5 Operações e Métodos Úteis.....	41
8.3.6 Dicionários Aninhados.....	42
8.4 Exercícios.....	43
CAPÍTULO 9: Leitura e Escrita de Arquivos CSV.....	45
9.1 O que são arquivos CSV?.....	45
9.2 Por que usar arquivos CSV?.....	45
9.3 Abrindo arquivos.....	46
9.4 Lendo um arquivo CSV.....	46
9.5 Escrevendo em um arquivo CSV.....	47
9.6 Exercícios.....	49
CAPÍTULO 10: Tratamento de Erros e Exceções.....	50
10.1 O que é o tratamento de erros e o que são exceções?.....	50
10.2 Por que tratar erros?.....	51
10.3 Como realizar o tratamento de erros em Python.....	51
10.4 Levantando exceções.....	52
10.5 Agrupando exceções.....	53
10.6 Exercícios.....	53
CAPÍTULO 11: A Teoria por trás da Orientação a Objetos.....	54
11.1 O que são classes e objetos.....	55
11.2 Vantagens da Orientação a Objetos.....	55
CAPÍTULO 12: Trabalhando com Classes e Objetos.....	57
12.1 Criando classes em Python.....	57
12.2 Encapsulamento, getters e setters.....	59
12.3 Métodos estáticos e métodos de classe.....	62
12.3.1 Métodos estáticos.....	62
12.3.2 Métodos de classe.....	63
12.3.3 Herança.....	63

12.3.4 Herança múltipla.....	67
12.3.5 Classes abstratas.....	69
12.3.6 Polimorfismo.....	71
12.3.7 Setinhas.....	74
CAPÍTULO 13: Automação.....	74
13.1 O que é automação de dados?.....	74
13.2 Por que automatizar dados?.....	75
13.3 Principais ferramentas e bibliotecas para a automação de dados.....	75
CAPÍTULO 14: Ciência de Dados.....	80
14.1 O que é ciência de dados?.....	80
14.2 Etapas do processo de ciência de dados.....	80
14.3 Principais ferramentas.....	81
14.4 Passo a passo com exemplos.....	82
CAPÍTULO 15: Desenvolvimento de Jogos.....	86
15.1 O que é Pygame?.....	86
15.2 Características do Pygame.....	87
15.3 Criando nosso primeiro jogo - Dodge the Blocks.....	90
15.3.1 Configuração Inicial.....	90
15.3.2 Criando o Jogador.....	91
15.3.3 Criando os Blocos Inimigos.....	91
15.3.4 Movimentando o Jogador.....	92
15.3.5 Movimentando os Blocos.....	93
15.3.6 Detectando Colisões.....	93
15.3.7 Desenhando os Elementos.....	94
15.3.8 Controlando a Velocidade do Jogo.....	94
15.4 Revivendo clássicos - Snake.....	95
15.4.1 Configuração Inicial.....	95
15.4.2 Criando a Cobra.....	96
15.4.3 Criando a Fruta.....	97
15.4.4 Movimentando a Cobra.....	97
15.4.5 Detectando Colisões.....	98
15.4.6 Comendo a Fruta.....	99
15.4.7 Atualizando o Corpo da Cobra.....	100
15.4.8 Desenhando os Elementos.....	100
15.4.9 Controlando a Velocidade.....	101
Considerações finais.....	102

Agradecimentos

Agradecimentos ao PET - Conexões de Saberes Ciência da Computação e à Profa. Dra. Alessandra Martins Coelho pela oportunidade e total colaboração durante todo o processo de desenvolvimento dessa apostila.

Agradecimentos especiais aos amigos Gabriel Fernandes e Wekisley Souza, da Ollie Academy, por toda a parceria durante a elaboração do material e a eterna amizade.

Obrigado por tudo. Todos serão inesquecíveis, e essa trajetória teria sido impossível sem a presença e a ajuda de vocês.

Eduardo Henrique

CAPÍTULO 1: Introdução

Olá! Sejam bem vindos ao curso de Introdução ao Python! Neste curso, você aprenderá os conceitos fundamentais da programação em Python, desde variáveis e estruturas de controle até funções e listas. Com exemplos práticos e exercícios, você desenvolverá habilidades essenciais para criar programas simples.

1.1 História do Python

Python é, sem dúvida, uma das linguagens de programação mais populares globalmente. Foi criada em 1991 por Guido van Rossum, um talentoso programador holandês, com o objetivo de oferecer uma experiência de aprendizado e uso simples.

A origem do nome "Python" é uma homenagem ao programa de televisão *"Monty Python 's Flying Circus"*, muito querido entre os programadores da época. Guido van Rossum buscava um nome curto, único e divertido, e "Python" acabou sendo a escolha perfeita.

Nos anos seguintes, a popularidade do Python cresceu rapidamente, principalmente entre desenvolvedores que buscavam uma linguagem fácil e acessível para criar scripts e programas de automação.

Com o passar do tempo, o Python se consolidou como uma das linguagens mais utilizadas em diversas áreas, como desenvolvimento web, ciência de dados, inteligência artificial e aprendizado de máquina.

A comunidade de desenvolvedores do Python tem sido um dos pilares de seu sucesso, com contribuições ativas e engajadas. Graças a essa comunidade, uma vasta biblioteca de módulos e pacotes foi desenvolvida, abrangendo desde a criação de gráficos até a análise de dados e a construção de aplicativos web.

Atualmente, o Python é uma das linguagens de programação mais valorizadas no mercado de trabalho, com gigantes da tecnologia, como Google, Amazon, Microsoft e Facebook, adotando-a em seus projetos. Muitos profissionais de tecnologia veem no Python uma opção de carreira promissora e empolgante.

Em resumo, a trajetória do Python é uma história de sucesso. Desde sua criação em 1991, a linguagem cresceu em popularidade e se consolidou como uma das mais importantes e valiosas em todo o mundo.

1.2 Paradigmas

O Python é uma linguagem de programação que suporta múltiplos paradigmas, o que significa que os desenvolvedores têm a flexibilidade de escolher o estilo de programação mais adequado para suas necessidades. Dentre os principais paradigmas suportados pelo Python, destacam-se o paradigma imperativo, o paradigma orientado a objetos e o paradigma funcional.

No paradigma **imperativo**, os programas são escritos em uma sequência de comandos que alteram o estado do programa. Os desenvolvedores podem utilizar estruturas de controle como loops e condicionais para controlar o fluxo do programa. Esse estilo de programação é especialmente útil para resolver problemas que exigem a execução de passos em ordem específica.

Já no paradigma **orientado a objetos (OO)**, os programas são estruturados em classes e objetos, permitindo a representação de entidades do mundo real e suas interações. Os objetos possuem atributos e métodos que definem seu comportamento. Esse paradigma favorece a reutilização de código e a organização estruturada dos programas, tornando-os mais modulares e fáceis de manter.

O paradigma **funcional**, por sua vez, enfatiza a utilização de funções puras, que não possuem efeitos colaterais e retornam um valor baseado apenas em seus argumentos. Esse estilo de programação promove a imutabilidade dos dados e o uso de funções de ordem superior, que podem receber outras funções como argumentos ou retorná-las como resultado. A programação funcional é particularmente eficaz na resolução de problemas complexos e na realização de operações em coleções de dados.

Além desses paradigmas, o Python também suporta o paradigma **procedural**, que é semelhante ao paradigma imperativo, mas enfatiza a decomposição do programa em procedimentos ou funções.

A flexibilidade do Python ao suportar múltiplos paradigmas é uma das razões para sua popularidade e ampla aplicabilidade em diferentes áreas. Os

desenvolvedores podem escolher o estilo de programação mais adequado para cada projeto, tornando o Python uma ferramenta poderosa e versátil para diversas tarefas, desde a criação de pequenos scripts até o desenvolvimento de sistemas complexos e aplicações de larga escala.

1.3 Classificação da linguagem quanto a ortografia

Python é uma linguagem de programação sensível a maiúsculas e minúsculas, o que significa que ela diferencia entre letras maiúsculas e minúsculas em nomes de variáveis, funções, palavras-chave e outros elementos de código. Isso significa que você não pode usar letras maiúsculas e minúsculas de forma intercambiável ao escrever código em Python. Por exemplo, as variáveis "minhaVariavel", "MinhaVariavel" e "MINHAVARIAVEL" seriam consideradas diferentes em Python. Isso torna a linguagem menos propensa a erros de digitação em comparação com linguagens que não são sensíveis.

1.4 Classificação da linguagem quanto a tipagem

Python é uma linguagem de programação de tipagem dinâmica e forte. Vamos entender o que isso significa:

Tipagem Dinâmica: Em Python, você não precisa declarar explicitamente o tipo de uma variável ao criá-la. O tipo da variável é determinado automaticamente com base no valor atribuído a ela. Além disso, você pode reatribuir valores de diferentes tipos a uma mesma variável ao longo do programa. Isso proporciona flexibilidade, mas também pode levar a erros difíceis de detectar se não for utilizado com cuidado.

Tipagem Forte: Em Python, as operações entre diferentes tipos de dados são restritas para evitar resultados inesperados ou ambíguos. A linguagem não realizará automaticamente conversões de tipo implícitas, a menos que sejam especificamente definidas para determinados tipos. Isso pode ajudar a evitar erros sutis e a tornar o código mais previsível.

1.5 Classificação da linguagem quanto a execução

Python é uma linguagem de programação semi-compilada. Isso significa que, em vez de traduzir todo o código para linguagem de máquina antes da execução, como é feito em linguagens compiladas, o Python interpreta e executa o código linha por linha em tempo de execução. O código-fonte Python é analisado, traduzido em bytecode intermediário e, em seguida, interpretado pela máquina virtual Python (PVM). Essa abordagem traz vantagens, como portabilidade e facilidade de desenvolvimento, mas também pode resultar em um desempenho ligeiramente inferior em comparação com linguagens compiladas.

1.6 Exercícios

- 1) Quem criou o Python? Qual era o intuito do(a) criador(a) ao desenvolver essa linguagem de programação?
- 2) Cite 3 áreas da computação onde o Python pode ser utilizado.
- 3) O nome “Python” foi escolhido em homenagem a:
 - a) Um animal;
 - b) Um programa de TV;
 - c) Uma bebida;
 - d) Um componente da CPU;
 - e) Uma piada interna entre os desenvolvedores.
- 4) Cite e explique 4 paradigmas de programação suportados pelo Python.
- 5) Assinale V para as alternativas verdadeiras e F para as falsas. Reescreva as falsas, de forma que elas sejam verdadeiras:
 - a) () Python é uma linguagem sensível a letras maiúsculas e minúsculas. Dessa forma, as variáveis ‘OLLIE’ e ‘ollie’ são consideradas diferentes.
 - b) () Python pode ser classificada como uma linguagem de tipagem dinâmica e fraca.
 - c) () Linguagens semi-compiladas, como Python, possuem um desempenho superior a linguagens compiladas.

CAPÍTULO 2: Ambiente de Desenvolvimento Windows

2.1 Objetivos

Nosso primeiro objetivo é aprender a instalar o Python. Durante essa etapa, você aprenderá a baixar e instalar a versão mais recente do Python em seu sistema operacional. Além disso, vamos abordar as configurações necessárias durante a instalação e garantir que tudo tenha sido concluído corretamente.

Em seguida, vamos focar na instalação do VS Code, um ambiente de desenvolvimento integrado (IDE) muito popular e eficiente. Você aprenderá como baixar e instalar o VS Code em seu sistema operacional e, em seguida, faremos as configurações básicas necessárias para iniciar o trabalho com o IDE.

Por fim, verificaremos se as instalações foram realizadas com sucesso. Faremos isso executando comandos Python no prompt de comando ou terminal para confirmar que o Python está funcionando corretamente. Além disso, verificaremos se o Python foi configurado corretamente nas variáveis de ambiente do sistema. Finalmente, criaremos um arquivo Python simples no VS Code e o executaremos para garantir que o editor esteja configurado corretamente.

Ao final dessa aula, você terá adquirido as habilidades necessárias para instalar o Python, o VS Code e verificar se as instalações foram concluídas com êxito. Esses passos são fundamentais para começar seus projetos de programação Python de forma eficiente e produtiva. Pratique e explore ainda mais essas ferramentas para aprimorar suas habilidades de programação Python. Boa sorte!

2.2 Instalação do Python

Para instalar o Python, você deve seguir algumas etapas simples. Primeiro, acesse o site oficial do Python (python.org) e baixe a versão mais recente do Python para o seu sistema operacional. Existem versões disponíveis para Windows, macOS e Linux, portanto, certifique-se de selecionar a correta.

Após baixar o instalador, execute-o e siga as instruções na tela. Durante o processo de instalação, você pode optar por adicionar o Python ao PATH do sistema, o que permite executar o Python facilmente a partir do prompt de comando ou terminal. Isso também facilita a execução de scripts Python de qualquer local do seu computador.

2.3 Instalação do VS Code

Primeiro, abra o seu navegador e acesse o site oficial do VSCode (<https://code.visualstudio.com>). Na página inicial do site, você encontrará o botão de download do VSCode. Clique nesse botão para fazer o download do instalador. Uma vez que o download estiver concluído, execute o arquivo de instalação do VSCode.

Na primeira tela do instalador, clique no botão “Next” para prosseguir com a instalação. Leia os termos e condições e concorde.

Na próxima tela, você pode escolher o local de instalação do VSCode. Se você estiver satisfeito com o local padrão, basta clicar em “Next”.


Em seguida, você pode optar por adicionar o VSCode ao PATH do sistema, o que permitirá que você abra o VSCode a partir do prompt de comando. Selecione a opção "Add to PATH" e clique em "Next".

Na tela seguinte, você pode escolher se deseja criar ícones de atalho na área de trabalho e no menu Iniciar. Selecione as opções desejadas e clique em “Next”.

Finalmente, clique em “Install” para iniciar a instalação do VSCode. Aguarde enquanto o instalador do aplicativo conclui a instalação. Quando a instalação estiver concluída, clique em "Finish" para sair do instalador.

Pronto! Agora você tem o VSCode instalado em seu computador e está pronto para começar a programar.

Se ainda restarem dúvidas quanto a esses passos, acesse o nosso canal do YouTube e acompanhe a instalação do Python e do VS Code passo a passo:

 **AULA 01 - Configurando o Ambiente de Desenvolvimento** . O link da aula é: <https://www.youtube.com/watch?v=PtvRUy4k-9U&t=6s> .

2.4 Configurações do VS Code

Nessa aula, algumas extensões são instaladas no VS Code. Para que possa fazer o mesmo e aproveitar o máximo de todo o aprendizado oferecido por nós, você pode repetir a instalação dessas extensões com os scripts das configurações usadas, e eles estão disponíveis na pasta “configs-vscode” do meu repositório do GitHub: <https://github.com/duduhenry/Ollie-Academy/blob/main/configs-vscode> .

CAPÍTULO 3: Variáveis e Tipos de Dados

3.1 Variáveis

Em programação, variáveis são espaços de memória reservados para armazenar valores. Elas são utilizadas para armazenar dados que podem ser manipulados ou utilizados em operações matemáticas, lógicas ou de comparação.

Para criar uma variável, é necessário definir um nome e um tipo, que representa o tipo de dado que será armazenado. Em muitas linguagens de programação, é necessário também definir um valor inicial para a variável.

3.2 Tipos de dados

Os tipos de dados definem o tipo de valor que pode ser armazenado em uma variável. Existem diversos tipos de dados, mas os mais comuns são:

- int: armazena valores numéricos inteiros, como 1, 2, 3, -4, -5, -6, etc.
- float: armazena valores numéricos com casas decimais, como 3.14, -2.5, 0.5, etc.
- str: variáveis de texto; usadas para armazenar sequências de caracteres, como palavras, frases ou até mesmo texto completo. Para criar uma variável de texto, você precisa colocar o valor entre aspas simples (' ') ou aspas duplas (" ")

- **boolean**: armazena valores lógicos, como verdadeiro (True) ou falso (False).

Existem ainda outros tipos de dados mais complexos, como datas, horários, caracteres especiais, entre outros.

3.3 Padrões de nomenclatura

1. **camelCase**
2. **snake_case** - (Python - variáveis, funções, métodos, módulos, pacotes)
3. **PascalCase** - (Python - Nomes de Classes)
4. **UPPER_CASE** - (Python - constantes)
5. **lowercase**
6. **kebab-case** - (html/CSS class)

3.4 Declarando variáveis

Como atribuir valores às variáveis? No Python, declarar variáveis é extremamente simples. Tudo o que precisamos é escolher um nome e atribuir um valor a ele. Para atribuir valores à uma variável, usamos o sinal de igualdade.

```
nome_variavel1 = 10
```

```
nome_variavel2 = 10.6
```

```
nome_variavel3 = 'declarando variaveis'
```

3.5 Dicas importantes

Nomenclatura Descritiva:

- Escolha nomes de variáveis que sejam descritivos e reflitam o propósito ou conteúdo da variável. Ao escolher um nome para uma variável, é bom escolher um nome que descreva bem o que vamos armazenar nela. Por exemplo, se vamos criar uma variável para armazenar o número de pessoas na festa, não é legal que coloquemos um nome como “x” ou “h”, embora seja mais fácil de escrever. Um exemplo de um nome melhor, seria

“numero_pessoas_festa” ou “pessoas_na_festa” se você preferir. Isso pode evitar confusões em códigos extensos ou até mesmo na manutenção.

Evitar Abreviações Excessivas:

- Evite abreviações que possam ser confusas ou difíceis de entender.
- Prefira clareza e legibilidade em vez de economizar alguns caracteres.

Consistência em Nomenclatura:

- Mantenha uma convenção de nomenclatura consistente em todo o código.
- Escolha entre camelCase, snake_case, PascalCase, etc., e aplique-o uniformemente.

Uso Adequado de Constantes:

- Use constantes para valores que não devem ser alterados durante a execução do programa. Isso torna o código mais legível e facilita a manutenção.
- Relembrando o padrão de nomenclatura de constantes: UPPER_CASE

3.6 Exercícios

1) Explique o que é uma variável em programação e por que ela é usada. Dê um exemplo simples de como você poderia usar uma variável para armazenar a idade de uma pessoa.

2) Liste e explique brevemente dois tipos de dados numéricos em Python. Dê um exemplo de cada tipo de dado e explique quando você usaria cada um deles.

3) De acordo com os padrões de nomenclatura no Python, conecte as letras às colunas.

- | | |
|---------------|--------------|
| a) camelCase | ()Funções |
| b) snake_case | ()Variáveis |
| c) PascalCase | ()Constante |
| d) UPPER_CASE | ()Classes |
| e) lowercase | |

Resolução: b-b-d-c

CAPÍTULO 4: Operações Aritméticas e Lógicas

Neste capítulo, aprenderemos todas as operações aritméticas e lógicas que podemos utilizar ao trabalhar com as variáveis. Essas operações são conceitos fundamentais que você precisa dominar para se tornar um programador eficiente. Além disso, veremos os conceitos básicos das operações aritméticas, como soma, subtração, multiplicação, divisão e módulo, e estudaremos os operadores lógicos para realizar comparações e obter valores booleanos.

4.1 Operações Aritméticas

As operações aritméticas são usadas para realizar cálculos matemáticos em Python. Sendo elas adição, Subtração, Multiplicação, Divisão, Módulo.

- Soma (+): O sinal de positivo, ou 'mais', é utilizado para somar uma variável a outra;
- Subtração (-): Em subtração é a mesma coisa, só alterando o sinal para o negativo, ou 'menos', para subtrair uma variável de outra;
- Multiplicação (*): Agora vamos multiplicar, para a multiplicação utilizamos o sinal de *;
- Divisão (/): E na divisão utilizamos o sinal da barra. É importante se atentar para não confundir essa barra com outras do teclado, como o 'pipe' (|) e a 'contrabarra' (\);
- Módulo (%): Além dessas 4, temos o mod, que retorna o resto da divisão entre dois números.

Um exemplo comum de utilização da operação de mod é verificar se um número é divisível por outro. Suponha que queremos verificar se um número é par. Podemos utilizar o operador de mod para isso. Por exemplo, se temos o número 10 e queremos saber se ele é par, podemos fazer a seguinte verificação:

```
numero = 10

if numero % 2 == 0:
    print("O número é par.")
else:
    print("O número é ímpar.")
```

Neste caso, o operador ‘%’ retorna o resto da divisão de **numero** por 2. Se o resultado for igual a zero, significa que o número é divisível por 2, ou seja, é par. Caso contrário, o número é ímpar.

Portanto, a utilização da operação de módulo nos permite realizar verificações específicas relacionadas à divisibilidade de números, o que pode ser útil em várias situações dentro da programação.

4.2 Operações Lógicas

As operações lógicas básicas em Python são as seguintes:

- Igualdade: `==`
- Desigualdade: `!=`
- Maior que: `>`
- Menor que: `<`
- Maior ou igual a: `>=`
- Menor ou igual a: `<=`
- Negação: `not`
- E lógico: `and`
- Ou lógico: `or`

Por exemplo, se quisermos verificar se um número é maior que outro em Python, podemos usar o operador `>`:

```
a = 10
b = 20
a_maior = a > b
print(a_maior)  # Isso imprimirá false, pois b é maior.
```


Além disso, podemos usar operações lógicas em conjunto com operações aritméticas para realizar cálculos mais complexos. Por exemplo:

```
a = 10
b = 20
c = 5
soma_e_maior = a + b > c
print(soma_e_maior) #Isso imprimirá true.
```

- Igualdade: ==

```
a = 10
b = 10
iguais = a == b
print(iguais)
#Isso imprimirá true.
```

Obs.: É importante lembrar que o sinal de igualdade simples significa **atribuição**! A igualdade entre variáveis é representada pelo sinal “igual duplo”.

- Desigualdade: !=

```
a = 10
b = 20
diferentes = a != b
print(diferentes)
#Isso imprimirá true.
```

- Maior que: >

```
a = 10
b = 20
maior = b > a
print(maior)
#Isso imprimirá true.
```

- Menor que: <

```
a = 10
b = 20
menor = a < b
print(menor)
#Isso imprimirá true
```

Obs.: O símbolo do 'jogo da velha' (#) indica um comentário, ou seja, a linha não atrapalha a execução do código.

- Maior ou igual a: `>=`

```
a = 10
b = 20
maior_igual = b >= a
print(maior_igual)
#Isso imprimirá true
```

- Menor ou igual a: `<=`

```
a = 10
b = 20
menor_igual = a <= b
print(menor_igual)
#Isso imprimirá true
```

- Negação: `not`

```
a = True
negacao = not a
print(negacao)
#Isso imprimirá false.
```

- E lógico: `and`

```
a = True
b = False
conjuncao = a and b
print(conjuncao)
#Isso imprimirá false
```

- Ou lógico: `or`

```
a = True
b = False
disjuncao = a or b
print(disjuncao)
#Isso imprimirá true
```

4.3 Operadores de Atribuição

Operadores de atribuição são geralmente usados para evitar repetição de código. Com tais operadores, não é necessário repetir o nome da variável. Eles combinam uma operação com uma atribuição, facilitando a manipulação e a atualização de valores. Neste capítulo, aprenderemos como funcionam e quando utilizar os principais operadores de atribuição em Python.

```
variavel = 0
# Incrementando sem operadores de atribuição:
variavel = variavel + 1
# Incrementando com operadores de atribuição:
variavel += 1
```

No exemplo acima podemos perceber que um operador de atribuição pode nos poupar de reescrever o nome da variável. Os **operadores de atribuição** no Python são:

- **+= (adicionar e atribuir)** → Este sinal é utilizado para somar um valor ao valor que já está na variável. O valor dessa soma é atribuído à variável.
- **-= (subtrair e atribuir)** → Este sinal é utilizado para subtrair um valor do valor que já está na variável.
- ***= (multiplicar e atribuir)** → Este sinal é utilizado para multiplicar o valor que já está na variável.
- **/= (dividir e atribuir)** → Este sinal é utilizado para dividir o valor que já está na variável.
- **%= (atribuir o resto)** → Este sinal é utilizado para substituir o valor que já está na variável pelo resto da divisão entre o valor da variável e um outro valor.
- ****= (atribuir o quadrado)** → Este sinal é utilizado para substituir o valor que já está na variável pelo valor na variável elevado a um outro valor.

Por exemplo, se quisermos adicionar 5 a uma variável `x` existente, podemos usar o operador `+=`:

```
x = 10
x += 5
print(x)
#Isso imprimirá 15.
```

Da mesma forma, se quisermos dividir uma variável `x` por `2` e atribuir o resultado a ela mesma, podemos usar o operador `/=`:

```
x = 10
x /= 2
print(x)
#Isso imprimirá 5.0.
```

Exemplos

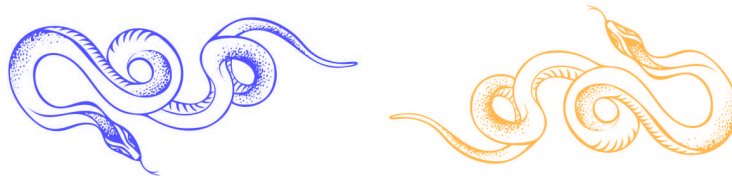
Veja abaixo alguns exemplos de como usar operadores de atribuição em Python:

```
# Atribuir valores a duas variáveis
x = 10
y = 5

# Usar operadores de atribuição para alterar valores das
variáveis
x += y # x agora é igual a 15
y *= 2 # y agora é igual a 10

# Imprimir os valores atualizados das variáveis
print(x) # Isso imprimirá 15
print(y) # Isso imprimirá 10
```

Bem, agora que chegamos ao final de mais um capítulo da nossa apostila, nada como finalizar com chave de ouro, né? Aqui vão alguns exercícios para que possamos praticar bastante e, dessa forma, aprendermos sempre mais, lembrando que as resoluções estão no nosso [GitHub](#). Tente fazer os exercícios sem olhar as respostas, pois ao olhar você estará prejudicando o seu próprio aprendizado. Boa sorte!

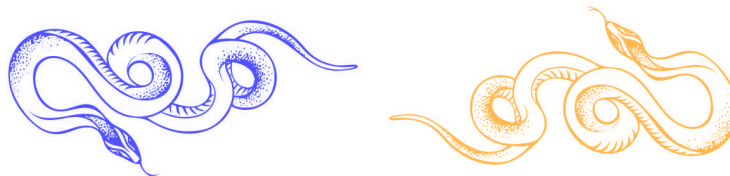


4.4 Exercícios

- 1) Escreva um programa que peça o preço de um produto e aplique um desconto de 10% usando o operador de atribuição adequado. Exiba o preço final com desconto.
- 2) Crie uma implementação que peça o valor de uma venda e calcule uma comissão de 5% sobre o valor. Use o operador de atribuição adequado para isso. Exiba o valor da comissão.
- 3) Escreva um código que peça ao usuário dois números e, em seguida, adicione o segundo número ao primeiro três vezes consecutivas, usando o operador de atribuição adequado. Exiba o valor final.
- 4) Escreva um programa que peça ao usuário a temperatura atual em graus Celsius. Aumente essa temperatura em 2 graus e depois subtraia 5 graus usando os operadores de atribuição adequados. Exiba o valor final da temperatura.
- 5) Crie um programa que peça ao usuário o tempo inicial de uma viagem (em horas). Depois, aumente o tempo da viagem em 3 horas usando o operador de atribuição adequado. Exiba o tempo total de viagem.
- 6) Peça ao usuário um número e divida-o por 2 usando o operador de atribuição adequado. Exiba o resultado final.

LEMBRETE: As respostas estão no nosso repositório do GitHub, que você pode acessar em <https://github.com/duduhenryy/Ollie-Academy> .

BONS ESTUDOS!



CAPÍTULO 5: Estruturas de Controle de Fluxo 'if', 'elif' e 'else'

Neste capítulo, estudaremos um pouco sobre algumas das mais usadas estruturas de controle de fluxo, que são `if` e `else`. Essas estruturas de controle de fluxo em Python permitem que o código execute diferentes ações com base em condições específicas.

- A estrutura `if` testa uma condição e executa um bloco de código se a condição for verdadeira. Por exemplo:

```
x = 10
if x > 5:
    print("x é maior que 5")
```

Nesse exemplo, o código testa se a variável `x` é maior que 5. Como `x` é igual a 10, a condição é verdadeira e o bloco de código abaixo do `if` é executado, imprimindo "x é maior que 5".

- A estrutura `else` é usada em conjunto com `if` e executa um bloco de código diferente se a condição for falsa. Por exemplo:

```
x = 2
if x > 5:
    print("x é maior que 5")
else:
    print("x é menor ou igual a 5")
```

Nesse exemplo, o código testa se a variável `x` é maior que 5. Como `x` é igual a 2, a condição é falsa e o bloco de código abaixo do `else` é executado, imprimindo "x é menor ou igual a 5".

- Também é possível usar a estrutura `elif` em conjunto com `if` para testar várias condições. Por exemplo:

```
x = 7
if x < 5:
    print("x é menor que 5")
elif x < 10:
    print("x é menor que 10, mas maior ou igual a 5")
else:
    print("x é maior ou igual a 10")
```

Nesse exemplo, o código testa se a variável `x` é menor que 5. Como `x` é igual a 7, a primeira condição é falsa. Em seguida, o código testa se `x` é menor que 10. Como `x` é menor que 10, mas maior ou igual a 5, a segunda condição é verdadeira e o bloco de código abaixo do `elif` é executado, imprimindo "x é menor que 10, mas maior ou igual a 5". Ou seja, o `elif` nada mais é que uma “mistura” entre o `if` e o `else`: `elif` + `if` = `elif`.

5.1 Importância da Indentação

Em Python, a indentação (espaços ou tabulações no início das linhas) é crucial. Ela define os blocos de código que pertencem às estruturas de controle de fluxo `if`, `elif` e `else`. Todos os comandos dentro de um bloco `if`, `elif` ou `else` devem estar igualmente indentados.

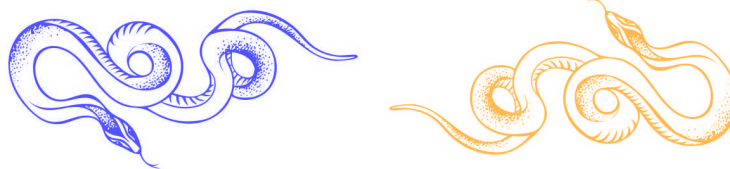
5.2 Exercícios

- 1) Faça um programa que avalie se um aluno foi aprovado na disciplina ou não, sabendo que as notas variam de 1 a 10 e que a média é 6.
- 2) Faça um programa que avalie se um número é positivo, negativo ou zero e escreva na tela a resposta.
- 3) Crie um programa que receba um número do usuário e printe `'False'` quando o número for par e `'True'` quando ímpar.
- 4) Peça dois números ao usuário e printe se o segundo número é divisível pelo primeiro.
- 5) Faça um programa que calcule o desconto em uma loja com base no preço do produto. E printe na tela o valor com o desconto aplicado.

- Comprou mais de 100 R\$ → 5% de desconto.
- Comprou mais de 300 R\$ → 10% de desconto.
- Comprou mais de 1000 R\$ → 15% de desconto.

6) Você foi contratado pelo Governo para criar um software de um radar de trânsito que ficará na rodovia de Cascavel - PR. O radar deve receber uma velocidade e, com base no valor, determinará se o veículo está respeitando o limite de velocidade de 80 km/h. Se o veículo estiver respeitando o valor de 80, o radar deve imprimir a mensagem de aceitação. Caso o veículo esteja acima desse limite, o radar automaticamente deve imprimir uma mensagem de multa e retornar o valor dessa infração com base no valor da porcentagem de excesso de velocidade atribuído ao valor padrão da multa de 120 reais.

Ex: se o veículo estiver a 100km/h, o valor da multa deverá ser computado como o valor padrão incrementado com o valor baseado na porcentagem de excesso, ou seja, de 20km/h.



CAPÍTULO 6: Estruturas de Controle de Fluxo ‘for’ e ‘while’

Em Python, as estruturas de controle de fluxo permitem que você controle a execução do código com base em determinadas condições. Duas das estruturas de controle de fluxo mais comuns são o loop `for` e o loop `while`. Essas estruturas são fundamentais para iterar sobre elementos em uma sequência ou repetir blocos de código enquanto uma condição for verdadeira. Vamos explorar cada uma delas em detalhes:

6.1 Loop 'for' em Python

O loop `for` é usado para iterar sobre uma sequência de elementos, como listas, tuplas, strings ou outros objetos iteráveis. A sintaxe geral do loop `for` é a seguinte:

```
for elemento in sequencia:
    # bloco de código a ser executado
```

O `elemento` é uma variável que assumirá os valores de cada elemento da `sequencia` em cada iteração do loop. O bloco de código dentro do loop será executado repetidamente, uma vez para cada elemento na sequência. Quando o loop termina, o controle é passado para a próxima instrução após o bloco `for`.

Exemplo : Iterar sobre uma lista

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

Nesse caso, será impresso na tela do usuário o nome de todas as frutas da lista, já que o loop `for` irá percorrer ela completamente até o final.

6.2 Loop 'while' em Python

O loop `while` é usado quando você deseja repetir um bloco de código enquanto uma determinada condição for verdadeira. A sintaxe geral do loop `while` é a seguinte:

```
while condição:
    # bloco de código a ser executado
```

O bloco de código será repetido enquanto a `condição` fornecida for avaliada como verdadeira. É importante ter cuidado ao usar o loop `while`, pois se a condição nunca for falsa, o loop pode executar indefinidamente, levando a um "loop infinito".

Exemplo: Contando até 5 usando um loop `while`

```
contador = 1
while contador <= 5:
    print(contador)
    contador += 1
```

Nesse cenário, o conteúdo a ser impresso na tela serão os valores computados pelo contador (neste caso os inteiros de 1 a 5). O loop `while` será executado enquanto a condição imposta for verdadeira, ou seja, enquanto a variável 'contador' possuir um valor menor ou igual a 5.

6.3 Exercícios

- 1) Utilizando o loop 'for', faça um programa que imprima todos os números pares de 1 a 20. Depois, faça outro código que imprima os ímpares.
- 2) Crie um programa que peça um número de 1 a 10 ao usuário e exiba a tabuada de multiplicação desse número.
- 3) Escreva um programa que some todos os números inteiros de 1 a 100 e exiba o resultado.
- 4) Desenvolva um programa em Python que use um loop `for` para imprimir os primeiros 10 números da sequência de Fibonacci. A sequência de Fibonacci é aquela em que cada número é a soma dos dois anteriores (começando com 0 e 1). Portanto, a sequência começa assim: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...
- 5) Utilize o loop `while` e escreva um programa que continue pedindo números ao usuário até que ele digite um número negativo, momento em que o programa deve terminar.
- 6) Escreva um programa que peça ao usuário um nome de usuário e senha. Dê ao usuário 3 tentativas para acertar o login. Após 3 tentativas erradas, exiba uma mensagem de erro e termine o programa.
- 7) DESAFIO: Crie um jogo simples em que o programa gera um número aleatório entre 1 e 10, e o usuário tem que adivinhar. O programa deve continuar pedindo palpites até o usuário acertar.

Pratique bastante para desenvolver sua lógica de programação. Resolva os exercícios com calma, sem consultar respostas, e só confira depois. BONS ESTUDOS!

CAPÍTULO 7: Funções

Em Python, funções são blocos de código reutilizáveis que podem ser chamados para realizar uma tarefa específica. Elas ajudam a modularizar o código, tornando-o mais organizado, legível e fácil de dar manutenção. As funções em Python seguem uma sintaxe específica e podem aceitar parâmetros de entrada e retornar valores. Aqui está a estrutura geral de uma função em Python:

```
def nome_da_funcao(parametro1, parametro2, ...):  
    # bloco de código da função  
    # pode incluir várias instruções  
    return valor_de_retorno
```

A definição de uma função começa com a palavra-chave **def**, seguida pelo **nome_da_funcao** (que você escolhe para identificar a função) e, entre parênteses, os parâmetros que a função pode receber (opcionais). O corpo da função é o bloco de código indentado logo abaixo da definição. O retorno é opcional, e quando uma função alcança uma instrução **return**, ela retorna o valor especificado. Se não houver instrução **return**, a função retornará **None** por padrão.

Vamos ver alguns exemplos. Nestes primeiros exemplos, veremos o funcionamento de algumas funções na prática e as coisas ficarão um pouco mais didáticas.

Exemplo 1: função que soma dois números;

```
def somar(a, b):  
    resultado = a + b  
    return resultado  
  
# Chamando a função e armazenando o resultado numa  
# variável  
  
resultado_soma = somar(5, 3)  
print(resultado_soma) # Output: 8
```

Exemplo 2: Função que verifica se um número é par

```
def verificar_par(numero):  
    if numero % 2 == 0:  
        return True  
    else:  
        return False  
# Chamando a função e imprimindo o resultado diretamente  
print(verificar_par(10)) # Output: True  
print(verificar_par(7)) # Output: False
```

Uma função pode ter qualquer número de parâmetros, e eles podem ser de tipos diferentes (inteiros, strings, listas, etc.). Além disso, você pode usar valores padrão para os parâmetros, tornando-os opcionais.

As funções são ferramentas poderosas para tornar o código Python mais modular, organizado e reutilizável. Elas permitem que você separe o código em blocos lógicos, tornando o processo de desenvolvimento mais eficiente e permitindo que você evite a repetição de código.

7.1 Funções úteis

int()

Converte um valor para inteiro. Exemplo:

```
num_inteiro = int("5")
```

str()

Converte um valor para string. Exemplo:

```
texto = str(42)
```

input()

Lê uma entrada do usuário. Exemplo:

```
nome = input("Digite seu nome: ")
```

len()

Retorna o tamanho de uma lista. Exemplo:

```
tamanho = len(numeros)
```

7.2 Exercícios

Agora que você está craque no assunto de criar funções, nada melhor que fazer alguns exercícios e praticar o aprendizado! Lembrando que todas as respostas estarão disponíveis no nosso [GitHub](#)!

- 1) Crie uma função que printe na tela uma mensagem de boas vindas.
- 2) Crie uma função chamada `saudacao` que receba o nome de uma pessoa como argumento e printe uma saudação personalizada. Por exemplo, se o nome fornecido for "Maria", a função deve printar "Olá, Maria!" .
- 3) Desenvolva uma função chamada `soma` que receba dois valores como argumento e retorne a soma dos dois, depois print na tela o resultado.
- 4) Crie uma função chamada `classifica_aluno` que receba a nota de um aluno em um exame como entrada. A função deve retornar a classificação do aluno com base na seguinte escala:
 - Nota maior ou igual a 9: "Excelente"
 - Nota entre 7 e 8.9: "Bom"
 - Nota entre 5 e 6.9: "Regular"
 - Nota abaixo de 5: "Insuficiente"

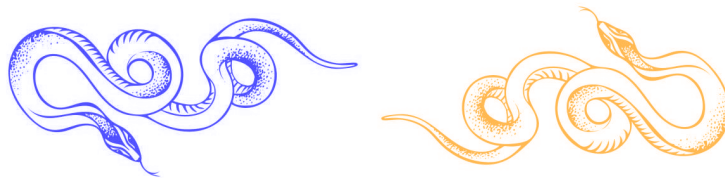
Além disso, se a nota for superior a 10 ou menor que 0, a função deve retornar "Nota inválida".

- 5) Crie uma função que receba dois números inteiros como parâmetro, esses valores serão informados pelo usuário, crie um laço que some do valor1 até o valor2, exemplo:

```
valor1 = 5  
valor2 = 10  
faça: 5+6+7+8+9+10
```

Extra:

Caso o valor1 seja maior que o valor2, não execute a função e printe uma mensagem de erro na tela "0 valor1 é maior que o valor2,", ou se caso os números forem iguais, informe uma mensagem de erro na tela "0 valor1 é igual o valor2,". Além disso, caso ele informe valores que se enquadram nesses dois casos, o programa não fecha, ele pede novamente os números.



CAPÍTULO 8: Listas, Tuplas e Dicionários

Agora, aprenderemos um pouco sobre tuplas, listas e dicionários. São estruturas de dados muito importantes e que possuem diversas aplicações e benefícios!

8.1 Listas

As listas em Python são estruturas de dados flexíveis que permitem armazenar múltiplos valores em uma única variável. Pense em uma lista como uma "caixa" que pode conter diferentes itens, como uma lista de compras. Por exemplo, imagine uma lista que armazena os itens de supermercado:

```
compras = ["maçãs", "leite", "pão"]
```

Além disso, você pode misturar diferentes tipos de itens na mesma lista. Pode ter números, palavras, até mesmo outras listas! Veja só:

```
misturado = [42, "olá", 3.14, ["uma", "outra", "lista"]]
```

Ah, e a ordem importa! Assim como na nossa lista de compras, a ordem dos itens na lista é importante. Agora, se quisermos acessar o que tem dentro da lista, usamos os índices de cada elemento. E lembrando: a contagem começa em zero!

```
print(misturado[1]) # Vai imprimir "olá"
print(compras[0])  # Vai imprimir "maçãs"
```

Um dos motivos principais para usar listas é sua capacidade de armazenar muitos valores sem a necessidade de criar várias variáveis. Por exemplo, em vez de criar mil variáveis para armazenar mil nomes, você pode usar uma única lista com todos eles. Agora, veremos algumas maneiras simples de manipular itens de listas.

8.1.1 Adicionando elementos nas listas

Adicionar elementos a uma lista é uma operação comum. Python oferece vários métodos para realizar essa tarefa:

Método `append()`:

O método `append()` é utilizado para adicionar um elemento ao final da lista.

```
frutas = ['maçã', 'banana', 'laranja']
frutas.append('uva')
print(frutas) #Saída: ['maçã', 'banana', 'laranja', 'uva']
```

Método `insert()`

Além do método `append()`, temos o método `insert()`, que permite adicionar um elemento em uma posição específica da lista.

```
numeros = [1, 2, 3, 5, 6]
numeros.insert(3, 4)
print(numeros) # Saída: [1, 2, 3, 4, 5, 6]
```

Veja que, passando o 3 e o 4 como parâmetro, o método `insert()` irá adicionar na lista, na posição do índice 3, o número quatro! Assim, os números que vêm à frente terão seus índices incrementados e continuarão na lista!

8.1.2 Removendo elementos das listas

Remover elementos de uma lista é outra operação importante. Python oferece métodos para remover elementos de diferentes maneiras:

Método `remove()`

O método `remove()` é usado para remover a primeira ocorrência de um elemento específico da lista.

```
animais = ['cachorro', 'gato', 'elefante', 'gato']
animais.remove('gato')
print(animais) # Saída: ['cachorro', 'elefante', 'gato']
```

Método `pop()`

O método `pop()` remove um elemento em uma posição específica da lista e retorna esse elemento.

```
cores = ['vermelho', 'azul', 'verde', 'amarelo']
cor_removida = cores.pop(1)
print(cor_removida) # Saída: 'azul'
print(cores) # Saída: ['vermelho', 'verde', 'amarelo']
```

8.1.3 Modificando elementos de uma lista

Modificar elementos em uma lista é útil para atualizar valores existentes:

```
notas = [85, 90, 78, 92]
notas[2] = 80
print(notas) # Saída: [85, 90, 80, 92]
```

8.1.4 Somando todos os números de uma lista

Para somar todos os números em uma lista basta usar a função `sum`:

```
notas = [85, 90, 78, 92]
soma_das_notas = sum(notas)
print(soma_das_notas) # Saída: 345
```


8.1.5 Descobrendo o tamanho de uma lista

Para descobrir o tamanho da lista, basta usar a função `len`:

```
notas = [85, 90, 78, 92]
numero_das_notas = len(notas)
print(numero_das_notas) # Saída: 4
```

8.1.6 Descobrendo o número de vezes que um elemento aparece na lista

Utilizamos a função `count()` para contar quantas vezes um elemento aparece na lista:

```
letras = ["a", "b", "a", "c", "a"]
print(letras.count("a")) # Saída: 3
```

8.1.7 Utilizando o loop for em listas

Quando combinados com listas em Python, os loops se tornam uma ferramenta poderosa para iterar sobre elementos e realizar diversas operações. Agora, vamos explorar como usar loops para percorrer listas, realizar ações em cada elemento e fornecer exemplos práticos para ilustrar esses conceitos.

O loop `for` é amplamente utilizado para iterar sobre os elementos de uma lista em Python. Ele executa um conjunto de instruções para cada elemento da lista.

```
frutas = ['maçã', 'banana', 'laranja']
for fruta in frutas:
    print(fruta)
# isso irá produzir: maçã
#                   banana
#                   laranja
```

8.1.8 Manipulando elementos com o loop for

Você pode realizar várias ações enquanto itera sobre os elementos de uma lista. Vamos ver algumas delas:

Exemplo: Somando Números em uma Lista

```
numeros = [1, 2, 3, 4, 5]
soma = 0
for numero in numeros:
    soma += numero
print("A soma dos números é:", soma) # 15
```

Exemplo: Modificando Elementos em uma Lista

```
nomes = ['Alice', 'Bob', 'Charlie']
for i in range(len(nomes)):
    nomes[i] = nomes[i].upper()
print(nomes) # ['ALICE', 'BOB', 'CHARLIE']
```

8.1.9 Usando a função enumerate() no loop for

A função `enumerate()` permite que você obtenha tanto o índice quanto o valor do elemento durante a iteração.

```
frutas = ['maçã', 'banana', 'laranja']
for indice, fruta in enumerate(frutas):
    print(f"Índice {indice}: {fruta}")
# Saída: Índice 0: maçã
#         Índice 1: banana
#         Índice 2: laranja
```

8.1.10 Usando while em listas

Além do loop `for`, você também pode usar o loop `while` para iterar sobre elementos de uma lista. Veremos um exemplo agora que remove os elementos de uma lista.

Exemplo: Removendo Elementos de uma Lista

```

numeros = [1, 2, 3, 4, 5, 3, 3, 3]
it = 0
while 3 in numeros:
    numeros.remove(3)
    it+=1
    print(f"iterações: {it}")
    print(numeros)
# SAÍDA: iterações: 1; lista atual: [1, 2, 4, 5, 3, 3, 3]
#         iterações: 2; lista atual: [1, 2, 4, 5, 3, 3]
#         iterações: 3; lista atual: [1, 2, 4, 5, 3]
#         iterações: 4; lista atual: [1, 2, 4, 5]

```

8.1.11 Exemplos Práticos de Loops com Listas

Vamos explorar alguns exemplos práticos que envolvem o uso de loops para iterar sobre elementos de uma lista:

Exemplo 1: Verificando Números Pares

```

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for numero in numeros:
    if numero % 2 == 0:
        print(numero, "é um número par")
    else:
        print(numero, "é um número ímpar")
# SAÍDA:
# 1 é um número ímpar
# 2 é um número par
# 3 é um número ímpar
# 4 é um número par
# 5 é um número ímpar
# 6 é um número par
# 7 é um número ímpar
# 8 é um número par
# 9 é um número ímpar
# 10 é um número par

```

Exemplo 2: Multiplicando Números em uma Lista

```

valores = [2, 3, 4, 5]
produto = 1
for valor in valores:
    produto *= valor
print("O produto dos valores é:", produto) # 120

```

Exemplo 3: Encontrando o Maior Valor

```
numeros = [17, 5, 42, 8, 23, 10]
maior = numeros[0]
for numero in numeros:
    if numero > maior:
        maior = numero
print("O maior número é:", maior) # 42
```

Loops com listas permitem que você processe e manipule elementos de forma eficaz. Eles são uma ferramenta fundamental para automatizar tarefas repetitivas e iterar sobre conjuntos de dados, tornando a programação mais eficiente e produtiva.

Agora que você já estudou bastante, vamos fazer milhares de exercícios! Brincadeira, antes de tudo, estudaremos as tuplas e, posteriormente, dicionários.

8.2 Tuplas

Tuplas são estruturas de dados em Python que permitem armazenar múltiplos valores em uma única variável, assim como as listas. A diferença principal é que as tuplas são **imutáveis**, ou seja, uma vez criadas, seus elementos não podem ser alterados, adicionados ou removidos. Isso as torna ideais para representar dados constantes ou informações que não devem ser modificadas durante a execução do programa.

Tuplas são criadas usando **parênteses** `()` ou simplesmente separando os valores por vírgulas. Por exemplo:

```
tupla = (1, 2, 3, "Python")
tupla_sem_parentheses = 1, 2, 3, "Olá"
```

8.2.1 Por quê usar tuplas?

1. Imutabilidade

A imutabilidade das tuplas garante que seus valores não sejam alterados acidentalmente durante a execução do programa. Isso é útil quando você

precisa de segurança em relação aos dados, como configurações, coordenadas ou valores constantes.

2. Eficiência

Tuplas geralmente consomem menos memória e são mais rápidas para serem processadas em comparação às listas. Isso as torna ideais para situações onde os dados são lidos com frequência, mas não precisam ser alterados.

3. Chaves em Dicionários

Como as tuplas são imutáveis, elas podem ser usadas como chaves em dicionários, ao contrário das listas, que são mutáveis e não podem ser usadas para esse propósito.

4. Desempacotamento Simples

Tuplas permitem que seus valores sejam desempacotados diretamente em variáveis, facilitando a manipulação de dados agrupados.

Os elementos de uma tupla podem ser acessados por seus índices, assim como nas listas. Lembre-se de que a contagem começa do zero. Tuplas também suportam índices negativos para acessar elementos a partir do final:

```
tupla = (10, 20, 30, 40)
print(tupla[0])    # Saída: 10
print(tupla[2])    # Saída: 30
print(tupla[-1])   # Saída: 40 (último elemento)
print(tupla[-2])   # Saída: 30 (penúltimo elemento)
```

8.2.1 Operações com Tuplas

Apesar de serem imutáveis, as tuplas suportam várias operações úteis e que podem ser utilizadas de formas simples.

Concatenar tuplas

Você pode combinar duas ou mais tuplas usando o operador `+`:

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
tupla3 = tupla1 + tupla2
print(tupla3)    # Saída: (1, 2, 3, 4, 5, 6)
```

Repetir tuplas

Multiplicar uma tupla por um número inteiro cria uma nova tupla com elementos repetidos:

```
tupla = (1, 2, 3)
print(tupla * 3)  # Saída: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Verificar existência de elementos

Podemos verificar a existência de um termo na tupla, a partir do uso do termo `in`

```
tupla = (10, 20, 30)
print(20 in tupla)  # Saída: True
print(40 in tupla)  # Saída: False
```

Iterar sobre tuplas

Assim como nas listas, você pode usar loops para iterar sobre os elementos:

```
tupla = ("a", "b", "c")
for letra in tupla:
    print(letra)
# SAÍDA:a
#       b
#       c
```

8.2.2 Desempacotamento de Tuplas

Tuplas permitem que seus valores sejam atribuídos diretamente a variáveis, facilitando a manipulação de dados agrupados:

```
tupla = (10, 20, 30)
a, b, c = tupla
print(a, b, c)  # Saída: 10 20 30
```

O próximo pseudocódigo exemplifica, também, a possibilidade de se ignorar valores em tuplas:

```
tupla = (1, 2, 3, 4)
a, _, c, _ = tupla  # Ignora os valores nos índices 1 e 3
print(a, c)         # Saída: 1 3
```

8.3 Dicionários

Dicionários são estruturas de dados em Python que armazenam informações no formato de **pares chave-valor**. Diferente das listas, onde os elementos são acessados por índices numéricos, nos dicionários os valores são acessados por meio de nomes. Isso torna os dicionários ideais para representar dados mais estruturados e organizados. Dessa forma, seria como se estivéssemos dando nomes aos índices.

Um exemplo de dicionário é o armazenamento de informações sobre um aluno:

```
aluno = {  
    "nome": "João",  
    "idade": 20,  
    "nota": 8.5  
}
```

Neste exemplo, `"nome"`, `"idade"` e `"nota"` são as chaves, enquanto `"João"`, `20` e `8.5` são os valores correspondentes.

8.3.1 Por quê usar dicionários?

1. **Acesso Rápido aos Dados** Com dicionários, é possível acessar informações diretamente por uma chave, sem precisar saber a posição do elemento, como acontece nas listas.
2. **Flexibilidade** Você pode armazenar diferentes tipos de dados (strings, números, listas, ou até mesmo outros dicionários) como valores.
3. **Organização de Dados Complexos** Dicionários permitem estruturar informações com múltiplos atributos em um único local.

8.3.2 Criando dicionários

Dicionários podem ser criados de várias formas:

Usando chaves: {}

```
dicionario_vazio = {}  
dicionario = {"nome": "Maria", "idade": 25}
```

Com a função `dict()`:

```
lista_de_tuplas = [("nome", "Ana"), ("idade", 22)]
dicionario = dict(lista_de_tuplas)
```

8.3.3 Acessando valores

Os valores de um dicionário podem ser acessados usando suas chaves:

```
aluno = {"nome": "João", "idade": 20, "nota": 8.5}
print(aluno["nome"]) # Saída: João
print(aluno["nota"]) # Saída: 8.5
```

Se você tentar acessar uma chave inexistente, o Python lançará um erro.

Para evitar isso, use o método `get()`:

```
print(aluno.get("curso", "Não especificado"))
# Saída: Não especificado
```

8.3.4 Modificando dicionários

Você pode adicionar, atualizar ou remover elementos de um dicionário facilmente:

Adicionando ou atualizando valores:

```
aluno["curso"] = "Engenharia"
aluno["idade"] = 21 # Atualiza o valor da chave "idade"
print(aluno) # Saída: {'nome': 'João', 'idade': 21,
                  'nota': 8.5, 'curso': 'Engenharia'}
```

Removendo Valores com o `del`:

```
del aluno["nota"]
print(aluno)
# Saída: {'nome': 'João', 'idade': 21, 'curso':
          'Engenharia'}
```

Removendo valores com o método `pop()`:

```
curso = aluno.pop("curso")
print(curso) # Saída: Engenharia
print(aluno) # Saída: {'nome': 'João', 'idade': 21}
```


Removendo todos os elementos com o método `clear()`:

```
aluno.clear()
print(aluno)    # Saída: {}
```

Removendo o último item

```
aluno = {"nome": "João", "idade": 20, "nota": 8.5}
aluno.popitem()
print(aluno)    # Saída: {'nome': 'João', 'idade': 20} (o
                último item foi removido)
```

8.3.4 Iterando sobre dicionários

Você pode percorrer um dicionário usando loops:

Percorrendo Apenas as Chaves

```
aluno = {"nome": "João", "idade": 20, "nota": 8.5}
for chave in aluno:
    print(chave)
# Saída: nome, idade, nota
```

Percorrendo Apenas os Valores

```
for valor in aluno.values():
    print(valor)
# Saída: João, 20, 8.5
```

Percorrendo Pares Chave-Valor

```
for chave, valor in aluno.items():
    print(f"{chave}: {valor}")
# Saída:
# nome: João
# idade: 20
# nota: 8.5
```

8.3.5 Operações e Métodos Úteis

Verificar se uma chave existe

```
print("nome" in aluno)    # Saída: True
print("curso" in aluno)   # Saída: False
```

Copiar um dicionário

```
copia = aluno.copy()
print(copia)  # Saída: {'nome': 'João', 'idade': 20,
                  'nota': 8.5}
```

Mesclar dicionários

Você pode usar o método `update()` para mesclar dois dicionários

```
info_adicional = {"curso": "Engenharia", "ano": 2023}
aluno.update(info_adicional)
print(aluno)  # Saída: {'nome': 'João', 'idade': 20,
                  'nota': 8.5, 'curso': 'Engenharia', 'ano': 2023}
```

Obter todas as chaves ou valores

```
print(aluno.keys())
# Saída: dict_keys(['nome', 'idade', 'nota', 'curso'])
print(aluno.values())
# Saída: dict_values(['João', 20, 8.5, 'Engenharia'])
```

8.3.6 Dicionários Aninhados

Dicionários podem conter outros dicionários, permitindo a criação de estruturas mais complexas:

```
alunos = { "aluno1": {"nome": "João", "idade": 20},
           "aluno2": {"nome": "Maria", "idade": 22} }

print(alunos["aluno1"]["nome"])  # Saída: João
```

Exemplo Prático

Imagine que você precisa armazenar uma lista de alunos com seus atributos (nome, idade e nota). Uma solução seria usar uma lista de dicionários:

```
lista_alunos = [
    {"nome": "Ana", "idade": 20, "nota": 8.5},
    {"nome": "Pedro", "idade": 22, "nota": 7.0}
]
for aluno in lista_alunos:
    print(f"Nome: {aluno['nome']}, Nota: {aluno['nota']}")
```

Agora sim, é hora de praticar!

Faça os exercícios propostos com atenção. Lembrando que as respostas estão no [GitHub](#). Depois de fazer tudo, tome a liberdade de conferir suas respostas por lá!

8.4 Exercícios

- 1) Crie uma lista chamada `compras` com os itens: "maçã", "banana" e "pão";
Adicione "leite" à lista;
Remova "pão" da lista;
Substitua "banana" por "laranja";
Imprima a lista final.
- 2) Crie uma lista com os números de 1 a 10 e use um loop para imprimir apenas os números pares.
- 3) Dada a lista `numeros = [10, 20, 30, 40, 50]`
Substitua o valor 30 por 35;
Adicione o número 60 ao final da lista;
Remova o número 10;
Imprima o tamanho da lista final.
- 4) Crie uma tupla chamada `coordenadas` que armazena os valores (4, 7).
Acesse e imprima os valores individuais da tupla.
- 5) Crie uma função que recebe dois números e retorna uma tupla com o quociente e o resto da divisão entre eles. Chame a função com os valores 17 e 5 e imprima o resultado.
- 6) Dada a tupla `dados = ("João", 25, "Engenharia")`:
Desempacote os valores em variáveis chamadas `nome`, `idade` e `curso`.
Imprima as variáveis individualmente.

7) Crie uma lista chamada `alunos`, onde cada elemento seja um dicionário representando um aluno com as seguintes informações: `nome`, `idade`, `nota`. Imprima o nome e a nota de cada aluno usando um loop.

8) Crie um dicionário para contar quantas vezes cada letra aparece na string "banana".

9) Crie um dicionário chamado `aluno` com as seguintes informações:

- Nome: "Maria"
- Idade: 22
- Nota: 8.5

Adicione o curso "Matemática" ao dicionário.

Atualize a nota para 9.0.

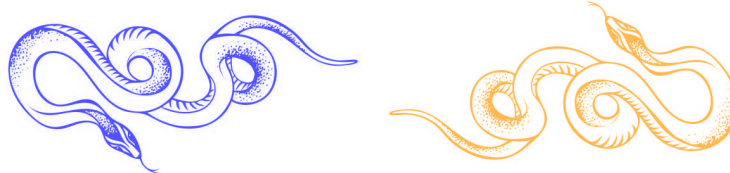
Remova a idade do dicionário.

Imprima o dicionário final.

10) Crie um dicionário chamado `precos` com os itens: "maçã" (2.50), "banana" (1.20), e "laranja" (3.00).

Pergunte ao usuário qual item ele deseja verificar.

Imprima o preço do item escolhido ou uma mensagem dizendo que o item não está disponível.



CAPÍTULO 9: Leitura e Escrita de Arquivos CSV

9.1 O que são arquivos CSV?

Os arquivos CSV (Comma-Separated Values) são amplamente utilizados para armazenar e transferir dados tabulares. Cada linha de um arquivo CSV representa um registro, e os valores de cada campo são separados por vírgulas. Eles são simples, fáceis de ler e suportados por muitas ferramentas, incluindo Python.

Geralmente cada coluna é separada por vírgula, mas isso pode variar de acordo com as convenções de outros lugares. Alguns separam as colunas com ponto e vírgula, outros com tabulação...enfim. Por exemplo, um arquivo CSV chamado `alunos.csv` pode conter:

```
nome,idade,nota
João,20,8.5
Maria,22,9.0
Pedro,19,7.5
```

Aqui, as colunas `nome`, `idade` e `nota` representam os cabeçalhos, enquanto as linhas abaixo contêm os registros.

9.2 Por que usar arquivos CSV?

Interoperabilidade

Arquivos CSV são compatíveis com diversas aplicações, como Excel e bancos de dados, facilitando o compartilhamento de informações.

Simplicidade

São fáceis de criar e manipular, mesmo com ferramentas básicas.

Leitura e Escrita Eficiente em Python

Python oferece bibliotecas integradas, como `csv`, que tornam a manipulação de arquivos CSV direta e eficiente.

9.3 Abrindo arquivos

Antes de manipular um arquivo CSV, é necessário abri-lo. Isso pode ser feito usando a função `open()`. Por exemplo:

```
arquivo = open("alunos.csv", "r")  # Abre o arquivo no
modo de leitura

conteudo = arquivo.read()          # Lê todo o conteúdo do
arquivo

print(conteudo)
arquivo.close()                    # Fecha o arquivo
```

É recomendado usar o contexto `with` para garantir que o arquivo seja fechado automaticamente, mesmo em caso de erros:

```
with open("alunos.csv", "r") as arquivo:
    conteudo = arquivo.read()
    print(conteudo)
```

9.4 Lendo um arquivo CSV

A biblioteca `csv` de Python simplifica a leitura de arquivos CSV. O método mais comum é usar o `csv.reader()`, que lê o arquivo linha por linha:

```
import csv
with open("alunos.csv", "r") as arquivo:
    leitor = csv.reader(arquivo)
    for linha in leitor:
        print(linha)
```

Exemplo de Saída:

```
['nome', 'idade', 'nota']
['João', '20', '8.5']
['Maria', '22', '9.0']
['Pedro', '19', '7.5']
```

Ignorando Cabeçalhos

Para ignorar os cabeçalhos, use a função `next()`:

```
with open("alunos.csv", "r") as arquivo:
    leitor = csv.reader(arquivo)
    next(leitor) # Ignora a primeira linha
    for linha in leitor:
        print(linha)
```

Acessando Campos Específicos

Como cada linha é uma lista, você pode acessar campos específicos por índice:

```
with open("alunos.csv", "r") as arquivo:
    leitor = csv.reader(arquivo)
    next(leitor) # Ignora os cabeçalhos
    for linha in leitor:
        print(f"Nome: {linha[0]}, Nota: {linha[2]}")
```

Lendo com DictReader

O `csv.DictReader` transforma cada linha em um dicionário, usando os cabeçalhos como chaves:

```
with open("alunos.csv", "r") as arquivo:
    leitor = csv.DictReader(arquivo)
    for linha in leitor:
        print(f"Nome: {linha['nome']}, Nota: {linha['nota']}")
```

Exemplo de Saída:

```
Nome: João, Nota: 8.5
Nome: Maria, Nota: 9.0
Nome: Pedro, Nota: 7.5
```

9.5 Escrevendo em um arquivo CSV

Para escrever em arquivos CSV, você pode usar o `csv.writer()` ou o `csv.DictWriter()`.

Usando `csv.writer()`

O método `writerow()`, intuitivamente traduzido, permite adicionar uma linha ao arquivo:

```
import csv
with open("novo_alunos.csv", "w", newline="") as arquivo:
    escritor = csv.writer(arquivo)
    escritor.writerow(["nome", "idade", "nota"]) # Cabeçalhos
    escritor.writerow(["Carlos", 21, 8.0])      # Primeira linha
    escritor.writerow(["Ana", 22, 9.5])         # Segunda linha
```

O resultado do código acima seria um arquivo csv contendo:

```
nome,idade,nota
Carlos,21,8.0
Ana,22,9.5
```

Escrevendo Múltiplas Linhas

Use o método `writerows()` para escrever várias linhas de uma vez:

```
dados = [
    ["nome", "idade", "nota"],
    ["Lucas", 20, 8.3],
    ["Fernanda", 23, 9.1]
]

with open("alunos_multiplas_linhas.csv", "w", newline="")
as arquivo:
    escritor = csv.writer(arquivo)
    escritor.writerows(dados)
```

Usando `DictWriter`

O `DictWriter` permite escrever dados diretamente de dicionários:

```
with open("alunos_dict.csv", "w", newline="") as arquivo:
    campos = ["nome", "idade", "nota"]
    escritor = csv.DictWriter(arquivo, fieldnames=campos)
    escritor.writeheader() # Escreve os cabeçalhos
    escritor.writerow({"nome": "Julia", "idade": 24, "nota":
8.7})
    escritor.writerow({"nome": "Marcos", "idade": 21, "nota":
7.9})
```


Modos de Abertura de Arquivos

Os modos de abertura determinam como o arquivo será manipulado:

- "r": leitura (padrão)
- "w": escrita (sobrescreve o conteúdo existente)
- "a": append (adiciona ao final do arquivo existente)
- "r+": leitura e escrita.

Exemplo:

```
with open("arquivo.csv", "a", newline="") as arquivo:
    escritor = csv.writer(arquivo)
    escritor.writerow(["Clara", 25, 9.2]) # Adiciona ao
                                         final
```

9.6 Exercícios

- 1) Dado o arquivo `produtos.csv` com o seguinte conteúdo:

```
Produto,Quantidade,Preco
Arroz,5,20.50
Feijão,3,8.90
Açúcar,2,4.50
```

Escreva um programa em Python que leia o arquivo e imprima cada linha no formato:

```
Produto: [Produto], Quantidade: [Quantidade], Preço: [Preco].
```

- 2) Crie um programa que escreva um arquivo `alunos.csv` com os seguintes dados:

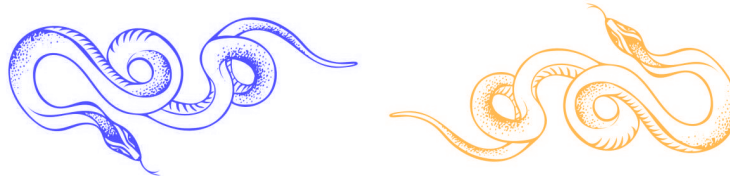
- Nome: Ana, Idade: 20, Nota: 8.5
- Nome: Pedro, Idade: 22, Nota: 7.0
- Nome: Carla, Idade: 19, Nota: 9.2

O arquivo deve conter os cabeçalhos `Nome`, `Idade` e `Nota`.

- 3) Dado o arquivo `produtos.csv` do **Exercício 1**, escreva um programa que leia o arquivo e calcule o valor total de cada produto (`Quantidade * Preço`). Imprima o resultado no formato:

```
Produto: [Produto], Valor Total: R$ [Valor Total].
```

- 4) Adicione os seguintes produtos ao final do arquivo `produtos.csv`:
- Produto: Leite, Quantidade: 10, Preço: 4.00
 - Produto: Café, Quantidade: 6, Preço: 12.50
- 5) Dado o arquivo `alunos.csv` gerado no **Exercício 2**, escreva um programa que:
- Calcule a média das notas dos alunos.
 - Identifique o aluno com a maior nota.
- 6) Usando o arquivo `alunos.csv`, escreva um programa que gere um novo arquivo chamado `aprovados.csv`, contendo apenas os alunos com nota maior ou igual a 8.0.



CAPÍTULO 10: Tratamento de Erros e Exceções

10.1 O que é o tratamento de erros e o que são exceções?

Durante a execução de um programa, erros podem ocorrer devido a diversos fatores, como entrada de dados inválida, arquivos inexistentes ou divisão por zero. Esses erros, chamados de exceções, interrompem a execução normal do programa, a menos que sejam tratados.

O tratamento de erros é um mecanismo que permite ao desenvolvedor prever e lidar com essas situações de forma controlada, garantindo que o programa continue funcionando adequadamente ou forneça mensagens de erro compreensíveis ao usuário.

10.2 Por que tratar erros?

Prevenir Quebras no Programa

Exceções não tratadas podem encerrar o programa inesperadamente. O tratamento de erros evita isso, permitindo que o código lide com situações inesperadas de forma segura, contornando-as e promovendo uma melhor experiência ao usuário.

Melhorar a Experiência do Usuário

Mensagens de erro claras ajudam o usuário a entender o que deu errado e como corrigir o problema.

Facilitar a Depuração

O tratamento adequado de exceções ajuda a identificar e isolar problemas no código, facilitando sua correção.

10.3 Como realizar o tratamento de erros em Python

O Python usa os blocos `try`, `except`, `else` e `finally` para tratar exceções de forma estruturada:

1. Bloco `try`

Contém o código que pode gerar uma exceção. O `try` literalmente tenta fazer algo. E se der algum erro nesse bloco, ele vai para o primeiro `except` que preveja esse erro.

```
try:
    #Código
```

2. Bloco `except`

Especifica o que deve ser feito se uma exceção for levantada no bloco `try`. Quando usamos esse `Exception`, ele pega qualquer erro. Mas é possível gerar erros mais específicos como os que vimos antes. Colocando `exception`, ele vai executar o bloco do `except` caso qualquer erro que aconteça.

```
try:
    # Código
except Exception:
```

Código

3. Bloco **else** (opcional)

Executado se nenhuma exceção for levantada. O bloco **else** é útil para executar código que só deve ser executado se não ocorrerem exceções no bloco **try**:

```
try:
    x = int(input("Dê um valor: "))
except ValueError as err:
    print("Deu ruim! ", err) # caiu na exceção
else:
    print("Deu bom!") # NÃO caiu na exceção
```

4. Bloco **finally** (opcional)

O bloco **finally** é sempre executado, mesmo que uma exceção ocorra. Ele é frequentemente usado para liberar recursos, como fechar arquivos:

```
try:
    arquivo = open("dados.txt", "r")
    conteudo = arquivo.read()
    print(conteudo)
except FileNotFoundError:
    print("Erro: O arquivo não foi encontrado.")
finally:
    print("Encerrando operação...")
    if 'arquivo' in locals() and not arquivo.closed:
        arquivo.close()
```

10.4 Levantando exceções

Você pode levantar suas próprias exceções usando a instrução **raise**. Isso é útil para validar entradas e interromper a execução se algo estiver errado:

```
def verificar_idade(idade):
    if idade < 18:
        raise ValueError("Idade mínima é 18 anos.")
    print("Acesso permitido.")

try:
    verificar_idade(16)
except ValueError as e:
    print(f"Erro: {e}")
```

A saída do programa acima será: Erro: Idade mínima é 18 anos.

10.5 Agrupando exceções

Várias exceções podem ser tratadas no mesmo bloco `except`, agrupando-as em uma tupla:

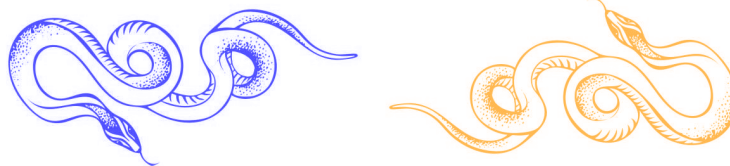
```
try:
    valor = int(input("Digite um número: "))
    resultado = 10 / valor
except (ZeroDivisionError, ValueError):
    print("Erro: Entrada inválida ou divisão por zero.")
```

Agora é hora de praticar!! Estude bem o conteúdo e faça os exercícios propostos. Depois de fazer tudo, sinta-se à vontade para ir no nosso [GitHub](#) para conferir as respostas no gabarito. Bons estudos!

10.6 Exercícios

- 1) Escreva um programa que receba dois números do usuário e calcule a divisão do primeiro pelo segundo. Trate os seguintes erros:
 - Divisão por zero.
 - Entrada inválida (quando o usuário digitar algo que não seja um número).
- 2) Escreva um programa que tente abrir um arquivo chamado `dados.txt` e exiba seu conteúdo. Trate o erro caso o arquivo não exista. Além disso, adicione um bloco `finally` para imprimir "Fim da execução", independentemente do resultado.
- 3) Escreva um programa que peça ao usuário para digitar um número e calcule a raiz quadrada desse número. Levante uma exceção personalizada se o número digitado for negativo.
- 4) Escreva uma função chamada `verificar_idade` que receba uma lista de idades e levante uma exceção para qualquer idade menor que 18. Use um `try` para capturar e exibir as exceções levantadas.

- 5) Escreva um programa que leia números de um arquivo chamado `numeros.txt`, some todos eles e exiba o resultado. Trate os seguintes erros:
- O arquivo não existe.
 - O arquivo contém valores inválidos.
 - O arquivo está vazio (levante uma exceção personalizada).
- Dica: use os métodos `readlines()` e `strip()`.
- 6) Escreva um programa que peça ao usuário para escolher um índice de uma lista e exiba o valor correspondente. Trate os seguintes erros:
- Índice fora do intervalo da lista.
 - Entrada inválida (quando o usuário não digitar um número).
- 7) Escreva um programa que implemente uma calculadora básica. O usuário deve informar dois números e uma operação (soma, subtração, multiplicação ou divisão). Trate os seguintes erros:
- Entrada inválida para os números.
 - Operação inválida.
 - Divisão por zero.



CAPÍTULO 11: A Teoria por trás da Orientação a Objetos

Primeiro de tudo, precisamos entender o que é essa tal de Orientação a objetos, que também podemos ver representada como OO.

A Orientação a Objetos é um paradigma de programação que se baseia na ideia de que o mundo pode ser modelado e representado como uma coleção de objetos, cada um com características (atributos) e comportamentos (métodos). Em

resumo, é uma abordagem que organiza o código em torno de "objetos" que interagem entre si.

11.1 O que são classes e objetos

Agora que entendemos o que é a orientação a objetos, vamos aprender sobre classes e objetos.

Classe é como um plano ou modelo que define as características (atributos) e ações (métodos) que nossos objetos terão. É como criar um esboço antes de desenhar uma figura.

Objeto é uma instância de uma classe. Eles são as coisas reais que criamos com base no plano da classe. Cada objeto tem seus próprios atributos e pode executar seus próprios métodos.

11.2 Vantagens da Orientação a Objetos

1. **Modelagem do Mundo Real:** A OO permite que você modele problemas de maneira semelhante à forma como entendemos o mundo real. Por exemplo, um sistema bancário pode ser modelado com objetos como "Conta", "Cliente" e "Transação", facilitando a compreensão e manutenção do código.
2. **Reutilização de Código:** A OO promove a reutilização de código. Uma vez que você cria uma classe (um modelo de objeto), você pode usá-la para criar vários objetos. Isso economiza tempo e evita duplicação de código.
3. **Organização e Estrutura:** A OO torna o código mais organizado e estruturado. Cada objeto tem um propósito definido e as relações entre objetos são claramente definidas, o que facilita a manutenção e a depuração.
4. **Abstração:** A OO permite a abstração, ou seja, você pode ocultar os detalhes de implementação complexos e se concentrar apenas na interface pública do objeto. Isso torna o código mais fácil de usar e entender.

5. **Encapsulamento:** O encapsulamento é um conceito-chave da OO que envolve a proteção dos atributos de um objeto e o acesso a eles. Isso permite controlar quem pode modificar os dados do objeto, tornando o código mais seguro.
6. **Herança:** A herança permite que você crie novas classes com base em classes existentes. Isso promove a reutilização de código e a criação de hierarquias de objetos, facilitando a modelagem de relacionamentos entre objetos.
7. **Herança múltipla:** A herança múltipla é um conceito de programação orientada a objetos que permite que uma classe herde características e comportamentos de várias classes pai. Com a herança múltipla, uma classe pode ter várias classes pai, o que significa que ela pode herdar atributos e métodos de todas essas classes. Isso permite que a classe filha tenha acesso a funcionalidades de diferentes classes, o que pode ser útil em situações em que uma classe precisa ter características de várias fontes diferentes.
8. **Polimorfismo:** O polimorfismo permite que objetos de diferentes classes respondam de maneira uniforme a chamadas de métodos comuns. Isso simplifica a interação entre objetos e torna o código mais flexível.
9. **Manutenção e Escalabilidade:** A OO facilita a manutenção do código à medida que ele cresce. Novas funcionalidades podem ser adicionadas criando novas classes ou modificando as existentes sem afetar o restante do sistema.

Em resumo, a Orientação a Objetos é uma abordagem poderosa para a programação que ajuda a tornar o código mais organizado, reutilizável e fácil de manter. É amplamente utilizada em muitas linguagens de programação, incluindo Python, para criar sistemas complexos e escaláveis.

Bom, agora que já discutimos e aprendemos o que é OO, daremos continuidade ao aprendizado de duas partes essenciais desse paradigma: classes e objetos.

CAPÍTULO 12: Trabalhando com Classes e Objetos

Neste capítulo, vamos explorar um dos conceitos da programação orientada a objetos: classes e objetos. Vamos aprender como criar classes, o que são objetos e como eles se relacionam. Imagine que estamos prestes a dar vida a personagens em nosso mundo de programação!

12.1 Criando classes em Python

Em Python, criar classes é bem simples. Por exemplo, vamos criar uma classe que represente uma pessoa:

```
class Pessoa:  
    . . .
```

Apenas com isso, já temos uma classe criada. Mas ela já consegue representar minimamente uma pessoa com isso? Não!

Para isso, temos mais alguns elementos, como os **atributos**. Mas, para atribuir atributos a uma pessoa, aqui, você precisa de um construtor. E pra criar um construtor:

```
class Pessoa:  
    # self é uma convenção  
    def __init__(self):  
        self.nome = None  
        self.idade = None
```

Com isso, você criou uma classe para representar uma pessoa. E essa classe define que uma pessoa tem um nome e uma idade. Se você precisar que o seu programa guarde CPF, telefone ou outras informações a mais, você pode colocar esses atributos também.

Apesar de você ter criado uma classe para representar uma pessoa, quando vamos fazer um código que use essa classe, criamos ela apenas com os dados que são úteis para o nosso programa, e não saímos criando dezenas de atributos para guardar todos os dados que se pode ter sobre uma pessoa. Para usar essa classe, é bem fácil também:

```
class Pessoa:
    def __init__(self):
        self.nome = None
        self.idade = None

if __name__ == '__main__':
    obj_pessoa = Pessoa()
```

No entanto, dessa forma, ainda não podemos criar um objeto pessoa com nome e idade. Para tornar obrigatória a criação de uma pessoa com esses dados preenchidos, podemos pedir isso no construtor:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = None
        self.idade = None

if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 34)
```

Assim, fica obrigatório informar o nome e a idade na hora de criar uma instância de “Pessoa”. Mas também é possível tornar opcional definindo um valor padrão como já fizemos com funções:

```
class Pessoa:
    def __init__(self, nome= 'Antonio', idade= 85):
        self.nome = None
        self.idade = None

if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 34)
```

Quem já viu OO em outras linguagens, pode ter pensado em sobrecarga quando viu essa questão de tornar os parâmetros opcionais. Então é bom saber que, no Python, não temos como aplicar a sobrecarga.

Para usar os atributos que definimos na classe:

```
class Pessoa:
    def __init__(self, nome= 'Antonio', idade= 85):
        self.nome = None
        self.idade = None

if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 34)
    print(obj_pessoa.nome) # a saída será João
```

Outra coisa que é possível fazer é definir ações, que são o que chamamos de **métodos**. Os métodos são bem parecidos com funções. O próprio construtor é um método. No caso, um método dunder.

Os métodos de uma classe geralmente definem ações que seriam responsabilidade dela fazer. Por exemplo, uma pessoa fala. Então seria plausível ver um método para falar na classe “Pessoa”. Mas uma pessoa não voa, então não seria plausível ter um método “voar” na classe “Pessoa”.

Então, para criar um método:

```
class Pessoa:
    def __init__(self, nome= 'Antonio', idade= 85):
        self.nome = None
        self.idade = None
    def fala(self):
        print("Olá mundo!")
if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 34)
    print(obj_pessoa.nome) # a saída será João
```

Se você quiser passar pro método o que a pessoa deve falar:

```
class Pessoa:
    def __init__(self, nome= 'Antonio', idade= 85):
        self.nome = None
        self.idade = None
    def fala(self, msg):
        print(msg)
if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 34)
    obj_pessoa.fala("Ola mundo!")
```

Caso você queira que o método retorne algo, é igual fazemos com funções também. Basta usar o “return”. Veremos isso agora em encapsulamento

12.2 Encapsulamento, *getters* e *setters*

Uma das vantagens de usar OO é o **encapsulamento**. Em outras linguagens, temos como definir quais classes vão poder acessar os atributos ou métodos da classe que estamos criando. Mas essa parte entenderemos melhor quando falarmos sobre herança. Existem três restrições: *public*, *protected* e *private*. No entanto, no

Python não temos essas restrições, mas temos convenções que tentam contornar isso.

O **public** usaremos quando quisermos que o atributo ou método possa ser acessado diretamente a partir de qualquer lugar fora da classe.

O **private** usaremos quando quisermos que um atributo ou método só seja acessado diretamente dentro da classe onde ele foi definido.

Para começar, não devemos deixar que alguém altere os atributos diretamente, então todos devem ser privados. Para fazer isso no python, colocamos dois underlines antes do nome do atributo:

```
class Pessoa:
    def __init__(self, nome='Anônimo', idade=0):
        self.__nome = nome
        self.__idade = idade

    def fala(self, msg):
        print(f'{self.__nome}: {msg}')

if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 84)
    obj_pessoa.fala('Olá!')
```

Caso precisar definir que um método só seja acessado diretamente a partir da classe onde ele foi definido, é só colocar os dois *underlines* antes do nome do método. Se quiser chamar o atributo diretamente, ainda é possível. Mas o programador que ver isso, quando tentar usar fora da classe, vai saber que está fazendo coisa errada.

Para acessarmos esses atributos ou alterarmos os mesmos, usamos os **getters** e os **setters**. Em outras linguagens, temos o padrão de apenas criar um método com *get* e depois o nome do atributo. Mas, no python, usamos *decorators* para isso. Com os decorators, não precisamos mudar a forma de chamar e alterar os atributos fora da classe.

Pra ficar mais claro, primeiro vamos começar vendo os *getters*:

```

class Pessoa:
    def __init__(self, nome='Anônimo', idade=0):
        self.__nome = nome
        self.__idade = idade

    @property
    def nome(self):
        return self.__nome

    def fala(self, msg):
        print(f'{self.__nome}: {msg}')

if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 84)
    print(obj_pessoa.nome)

```

Para fazer o *setter*:

```

class Pessoa:
    def __init__(self, nome='Anônimo', idade=0):
        self.__nome = nome
        self.__idade = idade

    @property
    def nome(self):
        return self.__nome
    @property
    def idade(self):
        return self.__idade
    @idade.setter
    def idade(self, idade):
        self.__idade = idade

    def fala(self, msg):
        print(f'{self.__nome}: {msg}')

if __name__ == '__main__':
    obj_pessoa = Pessoa('João', 84)
    obj_pessoa.idade = 12
    print(obj_pessoa.idade)

```

Caso você se pergunte o porquê de ter todo esse trabalho a mais, vou dar um exemplo. Esses *setters* podem impedir que a gente coloque qualquer coisa no atributo ou tratar um dado antes de colocá-lo no atributo. E isso pode diminuir a chance de acontecerem *bugs*. Um exemplo, a idade não pode ser negativa:

```

...
@idade.setter
def idade(self, idade):
    if idade >= 0:
        self.__idade = idade
    else:
        print("A idade atribuída é inválida!")
...

```

Com o `getter` é o mesmo. Você pode tratar um dado antes de retornar. Por exemplo: se você tem uma classe “Timer” que tem um atributo que guarda o tempo em milissegundos e quer que, quando alguém acesse o atributo, ele seja retornado em segundos. As aplicações vão depender do caso, mas são muito úteis.

12.3 Métodos estáticos e métodos de classe

12.3.1 Métodos estáticos

Para começar, precisamos entender o que são os métodos estáticos. Os métodos são, basicamente, funções dentro de uma classe. Você não precisa instanciar a classe pra usar esse método. Porém, esse método também não tem acesso aos atributos da classe.

Para criar um método estático:

```

class Gato:
    @staticmethod
    def mia():
        print("Miau!")

```

Para usar um método estático:

```

class Gato:
    @staticmethod
    def mia():
        print("Miau!")
if __name__ == '__main__':
    Gato.mia()

```

Caso necessário, ainda podemos passar e pedir parâmetros nesses métodos, além de retornar valores também.

12.3.2 Métodos de classe

Os métodos de classe são métodos que também podem ser usados sem instanciar a classe. Mas nesse caso, você passa para ele a própria classe, e não uma instância. Ou seja, ele não tem o atributo nome de um objeto que você criou. Ele tem apenas o “molde” dessa classe. Uma utilidade disso, são as **factories**.

As *factories*, como o próprio nome diz, são fábricas. No caso, são fábricas de objetos. Elas podem ser bem úteis em alguns casos.

Para criar um método de classe:

```
class Gato:
    @classmethod
    def mia(cls):
        print("Miau!")
if __name__ == '__main__':
    Gato.mia()
```

Exemplo prático:

```
class Gato:
    def __init__(self, nome, raca):
        self.nome = nome
        self.raca = raca

    @classmethod
    def cria_gato(cls, nome):
        return cls(nome, "Siamês")

if __name__ == '__main__':
    gato = Gato.cria_gato("Tom")
    print(gato.nome)
```

Podemos usar isso para criar vários objetos sem passar todos os parâmetros, caso um seja igual em todos. Poderia ser útil em determinados cenários.

12.3.3 Herança

Para começar, primeiro precisamos entender o que é a herança. A ideia base é simples: uma classe filha que herda métodos e atributos da classe pai. Por

exemplo: animal, gato e cachorro. Nesse exemplo, tanto o gato quanto o cachorro são animais, então as duas herdaram os métodos e atributos de animais

Na programação isso pode ir muito além, mas visualizar exemplos fica bem mais fácil de entender. Primeiro, vamos supor que temos uma classe “Pessoa” e uma classe “Cliente”:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

class Cliente:
    def __init__(self, nome, idade):
        ...
```

Pra fazer “Cliente” ser uma classe filha de “Pessoa”, é só colocar o nome da classe pessoa entre parênteses na frente de Cliente:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

class Cliente(Pessoa):
    def __init__(self, nome, idade):
        ...
```

Dessa forma, já é possível fazer a classe cliente herdar atributos e métodos da classe pessoa:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

class Cliente(Pessoa):
    def __init__(self, nome, idade):
        super().__init__(nome, idade)

if __name__ == '__main__':
    pessoa = Pessoa('Maria', 54)
    cliente = Cliente('João', 87)
    print(cliente.nome)
```


Outra coisa que legal, é poder aproveitar os métodos da classe mãe:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def fala_oi(self):
        print("Oi!")

class Cliente(Pessoa):
    def __init__(self, nome, idade):
        super().__init__(nome, idade)

if __name__ == '__main__':
    pessoa = Pessoa('Maria', 54)
    cliente = Cliente('João', 87)
    cliente.fala_oi()
```

O bom de usar isso é que não temos que refazer tudo o que fizemos antes. Então, podemos criar um comportamento uma vez em um método da classe mãe e apenas alterar o que precisamos na classe filha.

Caso você precise adicionar alguma coisa em um método da classe mãe, mas ainda precisar que esse método faça tudo que ele faz na classe mãe, também é possível. Por exemplo:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def fala_oi(self):
        print("Oi!")

class Cliente(Pessoa):
    def __init__(self, nome, idade):
        super().__init__(nome, idade)
    def fala_oi(self):
        super().fala_oi()
        print("Quero fazer mais!")

if __name__ == '__main__':
    pessoa = Pessoa('Maria', 54)
    cliente = Cliente('João', 87)
    cliente.fala_oi()
```

Protected

Não sei se vocês lembram, mas ficou combinado que só íamos falar de *protected* quando víssemos herança. Então, chegou a hora. Como já dissemos anteriormente, *private* só nos permite usar um atributo ou método na classe em que ele foi criado. O *public* deixa usar onde quiser. Agora, o *protected* só te deixa usar nas classes filhas da classe onde o atributo ou método foi criado, e nessa própria classe onde ele foi criado.

Então, a maneira certa de escrever os métodos da classe “Pessoa” seria:

```
class Pessoa:
    def __init__(self, nome, idade):
        self._nome = nome
        self._idade = idade

    def fala(self):
        print("IHUL!")

class Cliente(Pessoa):
    def __init__(self, nome, idade):
        super().__init__(nome, idade)

    def fala(self):
        super().fala()
        print("Quero fazer mais!")

if __name__ == '__main__':
    pessoa = Pessoa('Maria', 54)
    cliente = Cliente('João', 87)
    cliente.fala()
```

Dessa forma, só as classes filhas e a própria classe pessoa iria poder acessar esses métodos diretamente. Só pra relembrar:

- Sem underline → public
- Com uma underline → protected
- Com duas underlines → private

12.3.4 Herança múltipla

Além da herança normal, temos a possibilidade de fazer múltiplas heranças. Nesse caso, uma classe pode herdar de várias classes ao mesmo tempo. Mas é aconselhável não exagerar muito.

Para fazer uma classe herdar de mais de uma classe é bem fácil:

```
class Gato:
    ...
class Cachorro:
    ...
class BichoDeEstimacao(Cachorro, Gato):
    ...
if __name__ == '__main__':
    ...
```

Mas e se “Gato” e “Cachorro” fossem classes filhas de uma classe “Animal”? Teremos um problema que chamam de **problema do diamante**. Por exemplo:

```
class Animal():
    ...

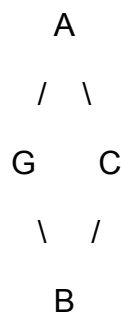
class Gato(Animal):
    ...

class Cachorro(Animal):
    ...

class BichoDeEstimacao(Gato, Cachorro):
    ...

if __name__ == '__main__':
    ...
```

Isso forma uma espécie de diamante com as heranças e classes:



Com isso, se for necessário chamar algum método de Animal, ele vai vir de “Gato”, ou de “Cachorro”? Em Python, existe uma ordem de precedência. Se você chama um método que existe em todas as classes do diamante, ele vai chamar o da base, depois o primeiro do meio, e ir seguindo em direção ao mais à direita. Depois ele chama o da ponta superior.

Pra ficar mais claro, vamos ver um exemplo:

```
class Animal():
    ...
    def metodo(self):
        print("Metodo Animal")

class Gato(Animal):
    ...
    def metodo(self):
        print("Metodo Gato")
        super().metodo()

class Cachorro(Animal):
    ...
    def metodo(self):
        print("Metodo Cachorro")
        super().metodo()

class BichoDeEstimacao(Gato, Cachorro):
    ...
    def metodo(self):
        print("Metodo Bicho de Estimação")
        super().metodo()

if __name__ == "__main__":
    bicho = BichoDeEstimacao()

    bicho.metodo()
```

Também é possível ver essa lista de precedência no Python usando um *dunder method*:

```
. . .
if __name__ == "__main__":
    bicho = BichoDeEstimacao()
    print(BichoDeEstimacao.__mro__) # retorna tupla
# OU
    print(BichoDeEstimacao.mro()) # retorna lista
```

Dessa forma, é como se estivéssemos chamando o super a partir de Gato, e não de bicho de estimação. Pode ser que seja útil em algum momento.

12.3.5 Classes abstratas

Para começar, temos que entender o que são classes abstratas. Podemos pensar nelas como um molde ou modelo para algo. Por exemplo, se vamos construir um carro, precisamos de um molde, um modelo. Um carro de um modelo A tem tais rodas, ele acelera de uma forma específica. Todos desse modelo serão assim. Então a gente já espera algumas características em comum entre carros de um mesmo modelo. Na programação, não é muito diferente.

Em Python, para poder usar as classes abstratas, precisamos importar da biblioteca abc a classe ABC. A nossa futura classe abstrata precisa ser filha da classe ABC. Por exemplo:

```
from abc import ABC

class Veiculo(ABC):
    def __init__(self, modelo):
        self.modelo = modelo

class Carro(Veiculo):
    def __init__(self, modelo):
        super().__init__(modelo)

if __name__ == '__main__':
    carro = Carro("XI3312")
```

Por enquanto, ela não faz nada diferente do que já vimos. Mas aqui vêm os métodos. As classes abstratas precisam ter ao menos um método abstrato. Sem um método abstrato, só temos herança ali. Para criar um método abstrato:

```
...
    @abstractmethod
    def acelerar(self):...

    @abstractmethod
    def freiar(self):...
...
```

```

if __name__ == '__main__':
    carro = Carro("XI3312")
    carro.acelerar()

```

Aqui, já começamos a ver a diferença de herança e abstração. Quando criamos métodos abstratos, se tornou obrigatório que os implementássemos nas classes filhas da classe abstrata.

```

...
class Carro(Veiculo):
    def __init__(self, modelo):
        super().__init__(modelo)

    def acelerar(self):
        print("Acelerando...")

    def freiar(self):
        print("Freiando...")

if __name__ == '__main__':
    carro = Carro("XI3312")
    carro.acelerar()
    carro.freiar()

```

Caso vocês tentarem criar os sets e gets, pode ser que tenham problemas. Então vejamos como fica:

```

...

    @abstractmethod
    def acelerar(self):...

    @abstractmethod
    def freiar(self):...

    @property
    @abstractmethod
    def modelo(self):...

    @modelo.setter
    @abstractmethod
    def modelo(self, modelo):...

class Carro(Veiculo):
    def __init__(self, modelo):
        super().__init__(modelo)

    def acelerar(self):
        print("Acelerando...")

```

```

        def freiar(self):
            print("Freiando...")
    ...

```

Agora, para implementar na classe filha:

```

from abc import ABC, abstractmethod

class Veiculo(ABC):
    def __init__(self, modelo):
        self._modelo = modelo
    ...
    @property
    @abstractmethod
    def modelo(self):
        return self._modelo

    @modelo.setter
    @abstractmethod
    def modelo(self, modelo):
        self._modelo == modelo

class Carro(Veiculo):
    ...
    @property
    def modelo(self):
        return self._modelo

    @modelo.setter
    def modelo(self, modelo):
        self._modelo == modelo

    def acelerar(self):
        print("Acelerando...")

    def freiar(self):
        print("Freiando...")

```

12.3.6 Polimorfismo

Agora, em relação ao polimorfismo, vamos perceber que é bem simples. Como o próprio nome já denuncia, polimorfismo se trata de múltiplas formas. Basicamente, vamos mostrar a possibilidade de uma função, método ou classe receber um parâmetro de uma forma(ou tipo) mais generalizado. Por exemplo, nós

temos uma classe veículo e temos um carro que pertence à classe veículo. Agora, vamos criar a classe moto:

```
from abc import ABC, abstractmethod

class Veiculo(ABC):
    def __init__(self, modelo):
        self._modelo = modelo

    @abstractmethod
    def acelerar(self):...

    @abstractmethod
    def freiar(self):...

    @property
    @abstractmethod
    def modelo(self):
        return self._modelo

    @modelo.setter
    @abstractmethod
    def modelo(self, modelo):
        self._modelo == modelo

class Carro(Veiculo):

    def __init__(self, modelo):
        super().__init__(modelo)

    @property
    def modelo(self):
        return self._modelo

    @modelo.setter
    def modelo(self, modelo):
        self._modelo == modelo

    def acelerar(self):
        print("Acelerando carro...")

    def freiar(self):
        print("Freiando...")

class Moto(Veiculo):
    def __init__(self, modelo):
        super().__init__(modelo)

    @property
    def modelo(self):
        return self._modelo
```



```

@modelo.setter
def modelo(self, modelo):
    self._modelo == modelo

def acelerar(self):
    print("Acelerando moto...")

def freiar(self):
    print("Freiando...")

if __name__ == '__main__':
    carro = Carro("XI3312")

    print(carro.modelo)

```

Com a classe moto, temos duas coisas que são um veículo: a moto e o carro. Agora entra o polimorfismo. Vamos supor que a gente tenha uma função que acelera um veículo:

```

from abc import ABC, abstractmethod

class Veiculo(ABC):
    ...

class Carro(Veiculo):
    ...

class Moto(Veiculo):
    ...

def acelera_veiculo(veiculo: Veiculo):
    veiculo.acelerar()

if __name__ == '__main__':
    carro = Carro("XI3312")
    acelera_veiculo(carro)

```

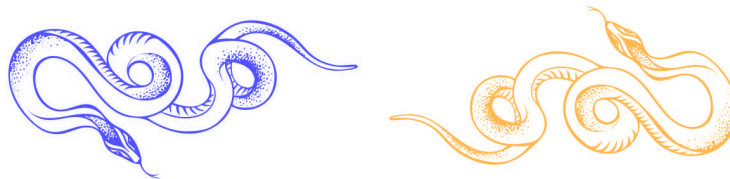
Nesse caso, especificamos que deveria ser passado um veículo no parâmetro, e pudemos passar o carro. Isso acontece porque definimos que o carro é um veículo, já que herdou as suas características. E como a moto também é um veículo, também podemos passar ela no parâmetro.

```
...  
  
if __name__ == '__main__':  
    moto = Moto("XI3312")  
    acelera_veiculo(moto)
```

12.3.7 Setinhas

Outra coisa que seria importante abordar são essas setinhas que o VS Code coloca no init. No Python, não temos como forçar qual vai ser o retorno de uma função ou método, mas temos como colocar um aviso. Se colocar uma seta na frente da assinatura e o tipo de retorno na frente, o VS Code vai avisar quando alguém for chamar a sua função ou método:

```
def retorna_int() -> int:  
    return 1
```



CAPÍTULO 13: Automação

13.1 O que é automação de dados?

Automação de dados é o processo de utilizar ferramentas e scripts para executar tarefas repetitivas relacionadas ao processamento, manipulação e análise de dados. Isso inclui a extração, transformação e carregamento de informações, permitindo que processos sejam realizados com menos intervenção manual, maior eficiência e menor propensão a erros.

Python é uma das linguagens mais utilizadas para automação de dados, graças à sua simplicidade e ao suporte de bibliotecas poderosas como `pandas`, `openpyxl`, `selenium` e `beautifulsoup`.

13.2 Por que automatizar dados?

1) Economia de Tempo

Reduz o tempo necessário para tarefas manuais repetitivas, como limpeza de dados, geração de relatórios e integração de sistemas.

2) Redução de Erros

Processos automatizados minimizam os erros humanos em operações complexas ou volumosas.

3) Escalabilidade

Scripts de automação podem lidar com grandes volumes de dados de maneira eficiente.

4) Integração de Sistemas

Conecta diferentes fontes de dados, como planilhas, APIs e bancos de dados, para consolidar informações.

13.3 Principais ferramentas e bibliotecas para a automação de dados

Pandas

A biblioteca `pandas` é uma das mais populares em Python para análise de dados. Ela fornece estruturas como **DataFrame** e **Series**, que facilitam a manipulação e organização de grandes volumes de dados.

Principais Funcionalidades:

- Leitura de arquivos CSV, Excel, JSON e SQL.
- Limpeza de dados, como remoção de valores ausentes e duplicados.
- Filtragem, agrupamento e agregação de dados.
- Geração de estatísticas descritivas.
- Integração com bibliotecas como **matplotlib** e **seaborn** para visualização de dados.

Exemplo de aplicação:

```
import pandas as pd

# Carregar dados de um arquivo CSV
df = pd.read_csv("dados.csv")

# Exibir os cinco primeiros registros
print(df.head())

# Filtrar registros onde a coluna "Idade" é maior que 30
filtro = df[df["Idade"] > 30]
print(filtro)
```

Openpyxl

openpyxl é uma biblioteca poderosa para manipular arquivos Excel em Python. Com ela, é possível criar, editar e formatar planilhas de maneira programática.

Principais Funcionalidades:

- Criação de novas planilhas e adição de dados.
- Leitura e edição de planilhas existentes.
- Aplicação de estilos, como negrito, cores e bordas.
- Inserção de gráficos diretamente no Excel.

Exemplo de aplicação:

```
from openpyxl import Workbook

# Criar uma nova planilha
wb = Workbook()
ws = wb.active
ws.title = "Planilha1"

# Adicionar dados
ws.append(["Nome", "Idade", "Nota"])
ws.append(["João", 25, 8.5])
ws.append(["Maria", 22, 9.0])

# Salvar o arquivo
wb.save("dados.xlsx")
```

Selenium

selenium é uma biblioteca usada para automatizar interações em navegadores da web. É amplamente utilizada para testes de interface, extração de dados de páginas dinâmicas e automação de tarefas online, como login e preenchimento de formulários.

Principais Funcionalidades:

- Abertura de páginas web e interação com elementos, como botões e campos de texto.
- Extração de informações de páginas dinâmicas.
- Suporte a múltiplos navegadores, como Chrome, Firefox e Edge.

Exemplo de aplicação:

```
from selenium import webdriver

# Abrir um navegador
driver = webdriver.Chrome()

# Navegar para uma página
driver.get("https://example.com")
```

```
# Localizar e preencher um campo de texto
campo_busca = driver.find_element("id", "campo_busca")
campo_busca.send_keys("Python")

# Fechar o navegador
driver.quit()
```

Beautifulsoup

beautifulsoup é uma biblioteca poderosa para análise de HTML e extração de dados de páginas web. Ela é ideal para web scraping, permitindo navegar pela estrutura do DOM e localizar elementos específicos.

Principais Funcionalidades:

- Localização de elementos HTML com seletores CSS ou tags.
- Extração de tabelas, listas e links.
- Navegação em estruturas de HTML complexas.

Exemplo de aplicação:

```
from bs4 import BeautifulSoup
import requests

# Fazer uma requisição para uma página web
url = "https://example.com"
resposta = requests.get(url)

# Analisar o HTML da página
soup = BeautifulSoup(resposta.text, "html.parser")

# Extrair e exibir todos os títulos <h1>
titulos = soup.find_all("h1")
for titulo in titulos:
    print(titulo.text)
```

Schedule e time

`schedule` é uma biblioteca leve usada para agendar a execução de tarefas em intervalos regulares. Junto com `time`, ela permite criar automações que rodam continuamente no fundo, como verificações periódicas ou atualizações automáticas.

Principais Funcionalidades:

- Agendamento de tarefas por segundo, minuto, hora ou dia.
- Execução de funções repetidamente com intervalos definidos.

Exemplo de aplicação:

```
import schedule
import time

def tarefa():
    print("Executando tarefa...")

# Agendar para executar a cada 10 segundos
schedule.every(10).seconds.do(tarefa)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Comparação Geral

Biblioteca	Finalidade	Ideal Para
<code>pandas</code>	Manipulação de dados tabulares	Limpeza, análise e processamento de grandes volumes de dados em arquivos CSV ou Excel.
<code>openpyxl</code>	Manipulação de arquivos Excel	Leitura, escrita e formatação de planilhas Excel.
<code>selenium</code>	Automação de navegação web	Interações com páginas dinâmicas e automação de tarefas como login e testes de interface.
<code>beautifulsoup</code>	Extração de dados de páginas estáticas	Web scraping e extração de informações específicas de páginas HTML.
<code>schedule/time</code>	Automação com base em horários	Execução de tarefas em intervalos regulares, como backups ou verificações automáticas.

CAPÍTULO 14: Ciência de Dados

14.1 O que é ciência de dados?

Ciência de Dados é o campo que combina conhecimentos de estatística, matemática, computação e áreas específicas de domínio para extrair informações valiosas a partir de dados. Envolve a coleta, organização, análise e visualização de grandes volumes de informações para apoiar a tomada de decisões.

Esse processo abrange desde a manipulação de dados brutos até a criação de modelos preditivos usando técnicas de aprendizado de máquina. Python é uma das linguagens mais utilizadas para ciência de dados, devido às suas bibliotecas e ferramentas robustas.

14.2 Etapas do processo de ciência de dados

Coleta de Dados

A primeira etapa envolve a obtenção de dados de várias fontes, como bancos de dados, APIs, arquivos CSV, ou extração direta da web (web scraping).

Limpeza e Preparação de Dados

Os dados brutos geralmente contêm erros, valores ausentes ou inconsistências. Nesta etapa, são aplicadas técnicas para organizar, padronizar e limpar os dados.

Exploração de Dados (EDA - Exploratory Data Analysis)

Compreender os dados é fundamental. Isso inclui análise estatística, identificação de padrões e relações entre variáveis, e criação de visualizações.

Modelagem e Machine Learning

Nessa fase, são construídos modelos preditivos ou descritivos para extrair insights e gerar previsões a partir dos dados.

Comunicação e Visualização de Dados

Por fim, os resultados são apresentados por meio de gráficos e relatórios que tornam as informações acessíveis e compreensíveis.

14.3 Principais ferramentas

1 - Manipulação de Dados

- **pandas**: Estruturas de dados como DataFrame para manipulação tabular.
- **numpy**: Operações matemáticas e manipulação de arrays.

2 - Visualização de Dados

- **matplotlib**: Gráficos simples e personalizáveis.
- **seaborn**: Visualizações estatísticas elegantes e informativas.

3 - Machine Learning

- **scikit-learn**: Modelos preditivos e ferramentas para aprendizado supervisionado e não supervisionado.

4 - Big Data e Bancos de Dados

- **pyspark**: Processamento de dados em grande escala.
- **SQLAlchemy**: Conexão com bancos de dados SQL.

5 - Outras Ferramentas

- **statsmodels**: Análise estatística e econometria.
- **tensorflow e pytorch**: Redes neurais e aprendizado profundo (Deep Learning).

14.4 Passo a passo com exemplos

1. Coleta de Dados

Lendo Dados de um Arquivo CSV

```
import pandas as pd

df = pd.read_csv("dados.csv")
print(df.head()) # mostra as 5 primeiras linhas do
arquivo
```

Extraindo Dados de uma API

```
import requests

resposta = requests.get("https://api.example.com/dados")
dados = resposta.json() # Converte os dados JSON em um
dicionário
print(dados)
```

2. Limpeza e Preparação de Dados

Removendo Valores Ausentes

```
df = df.dropna() # Remove linhas com valores ausentes
```

Substituindo Valores Faltantes por uma Média

```
df["coluna"] = df["coluna"].fillna(df["coluna"].mean())
```

Padronizando Dados

```
df["nome"] = df["nome"].str.lower()
# Converte textos para letras minúsculas
```

3. Exploração de Dados

Estatísticas Básicas

```
print(df.describe())  
# Estatísticas descritivas para cada coluna numérica
```

Correlação Entre Variáveis

```
print(df.corr())  
# Matriz de correlação entre variáveis numéricas
```

Visualizando Dados com Histogramas

```
import matplotlib.pyplot as plt  
  
df["coluna"].hist(bins=20)  
plt.show()
```

4. Modelagem e Machine Learning

Dividindo os Dados em Treinamento e Teste

```
from sklearn.model_selection import train_test_split  
  
X = df[["coluna1", "coluna2"]] # Variáveis independentes  
y = df["coluna_alvo"]         # Variável dependente  
X_train, X_test, y_train, y_test = train_test_split(X, y,  
test_size=0.2, random_state=42) # normalmente a divisão é  
80/20 para treino e teste, para que o modelo treine mais
```

Treinando um Modelo de Regressão Linear

```
from sklearn.linear_model import LinearRegression  
  
modelo = LinearRegression()  
modelo.fit(X_train, y_train) # Treina o modelo  
previsoes = modelo.predict(X_test)  
print(previsoes)
```

Avaliando o Modelo

```
from sklearn.metrics import mean_squared_error

erro = mean_squared_error(y_test, previsoes)
print(f"Erro Quadrático Médio: {erro}")
```

5. Visualização de Dados

Gráfico de Dispersão com **seaborn**

```
import seaborn as sns

sns.scatterplot(data=df, x="coluna1", y="coluna2")
plt.show()
```

Gráfico de Barras

```
df["categoria"].value_counts().plot(kind="bar")
plt.show()
```

A ciência de dados é uma área interdisciplinar que combina estatística, matemática e computação para extrair informações valiosas de grandes volumes de dados. Utilizando Python e suas ferramentas poderosas, os cientistas de dados coletam, limpam, analisam e visualizam dados, transformando informações complexas em insights úteis para a tomada de decisões.

O processo de ciência de dados segue etapas claras: desde a coleta e preparação dos dados até a modelagem com aprendizado de máquina e a apresentação dos resultados por meio de gráficos e relatórios. Ferramentas como **pandas** e **numpy** ajudam na manipulação dos dados, enquanto bibliotecas como **matplotlib** e **scikit-learn** permitem criar visualizações e modelos preditivos.

Por exemplo, podemos ler arquivos CSV com **pandas**, limpar dados ausentes substituindo por médias, explorar correlações entre variáveis e treinar modelos de regressão para previsões. Além disso, gráficos tornam os resultados mais compreensíveis, tornando a ciência de dados um pilar essencial na análise moderna.

A prática constante com exemplos e experimentos é fundamental para dominar essa área fascinante. Dessa forma, vamos ver um exemplo prático completo cujo objetivo é analisar dados de vendas de produtos e prever o total de vendas.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# 1. Carregar os Dados
df = pd.read_csv("vendas.csv")

# 2. Limpeza e Preparação
df = df.dropna() # Remover linhas com valores ausentes
df["Total"] = df["Quantidade"] * df["Preço"] # Criar coluna de total

# 3. Exploração
print(df.describe())
df["Total"].hist(bins=10)
plt.show()

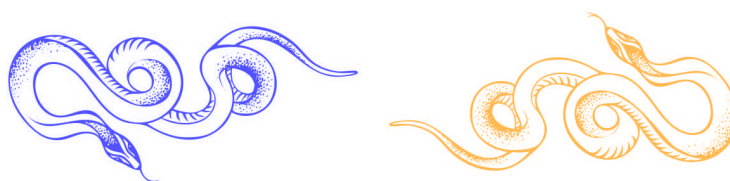
# 4. Modelagem
X = df[["Quantidade", "Preço"]] # Variáveis independentes
y = df["Total"] # Variável dependente
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

modelo = LinearRegression()
modelo.fit(X_train, y_train)

# 5. Avaliação
previsoes = modelo.predict(X_test)
erro = mean_squared_error(y_test, previsoes)
print(f"Erro Quadrático Médio: {erro:.2f}")

# Visualizar as Previsões
plt.scatter(y_test, previsoes)
plt.xlabel("Valores Reais")
plt.ylabel("Previsões")
plt.show()
```

A ciência de dados é um campo poderoso e dinâmico, essencial para transformar dados em insights acionáveis. Agora é sua vez! Com base no que aprendeu neste capítulo, procure uma base de dados na internet (em sites como o kaggle) e exercite todos os conceitos que aprendeu aqui. Lembre-se de estudar com calma e atenção!



CAPÍTULO 15: Desenvolvimento de Jogos

Chegamos ao último (e talvez o mais esperado) capítulo da nossa apostila de introdução ao Python: falaremos sobre criação de jogos! Usaremos a biblioteca Pygame, que oferece um grande suporte nessa área de desenvolvimento, que desperta grande interesse em diversos programadores.

15.1 O que é Pygame?

Pygame é uma biblioteca de Python voltada para a criação de jogos e outras aplicações multimídia. Ela fornece ferramentas para manipulação de gráficos, sons e interações com o usuário de forma simples e eficiente. Pygame é amplamente utilizado por desenvolvedores iniciantes e avançados devido à sua flexibilidade e facilidade de uso. Para instalá-lo, basta digitar no terminal o seguinte comando:

```
pip install pygame.
```

15.2 Características do Pygame

Gráficos 2D

Permite criar e manipular elementos gráficos em 2D, como sprites, imagens e animações.

Suporte a Áudio

Reproduz efeitos sonoros e músicas com facilidade.

Interação com o Usuário

Detecta eventos do teclado, mouse e até mesmo dispositivos de entrada como joysticks.

Compatibilidade Multiplataforma

Funciona em Windows, macOS e Linux, tornando-o acessível para diversos públicos.

Estrutura Básica de um Jogo com Pygame

Um jogo em Pygame geralmente segue estas etapas:

- Inicializar o Pygame e seus módulos.
- Criar a janela principal do jogo.
- Configurar variáveis e elementos do jogo.
- Iniciar o loop principal, que inclui:
 - ◆ Capturar eventos (teclado, mouse, etc.).
 - ◆ Atualizar o estado do jogo.
 - ◆ Desenhar os elementos na tela.
- Encerrar o jogo quando necessário.

Exemplo Básico: Janela do Jogo

Aqui está um exemplo simples de como criar uma janela básica em Pygame:

```

import pygame
import sys

# Inicializar o Pygame
pygame.init()

# Configurações da janela
largura, altura = 800, 600
tela = pygame.display.set_mode((largura, altura))
pygame.display.set_caption("Meu Primeiro Jogo")

# Loop principal
while True:
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT: # Fechar a janela
            pygame.quit()
            sys.exit()

    # Preencher o fundo com uma cor
    tela.fill((0, 0, 255)) # Azul
    pygame.display.flip() # Atualizar a tela

```

Adicionando Elementos ao Jogo

1. Desenhando Formas Básicas

Pygame permite desenhar formas geométricas, como retângulos e círculos, diretamente na tela.

Exemplo:

```

# Desenhar um círculo
pygame.draw.circle(tela, (255, 0, 0), (400, 300), 50) #
Vermelho, no centro da tela

```

2. Exibindo Imagens

Exemplo:

```

# Carregar e exibir uma imagem
imagem = pygame.image.load("jogador.png")
tela.blit(imagem, (100, 100)) # Exibe a imagem na posição
(100, 100)

```


3. Movendo Objetos

Movimento é um dos elementos mais importantes em jogos.

Exemplo:

```
x, y = 400, 300 # Posição inicial
velocidade = 5

while True:
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # Movimentar o objeto com as teclas de seta
    teclas = pygame.key.get_pressed()
    if teclas[pygame.K_UP]:
        y -= velocidade
    if teclas[pygame.K_DOWN]:
        y += velocidade
    if teclas[pygame.K_LEFT]:
        x -= velocidade
    if teclas[pygame.K_RIGHT]:
        x += velocidade

    # Atualizar a tela
    tela.fill((0, 0, 0)) # Fundo preto
    pygame.draw.circle(tela, (255, 0, 0), (x, y), 50) #
    Desenhar o círculo
    pygame.display.flip()
```

Adicionando Som

Sons tornam os jogos mais envolventes. Pygame suporta efeitos sonoros (`pygame.mixer.Sound`) e música de fundo (`pygame.mixer.music`).

```
# Inicializar o mixer
pygame.mixer.init()

# Carregar e tocar uma música de fundo
pygame.mixer.music.load("musica_fundo.mp3")
pygame.mixer.music.play(-1) # Tocar em loop

# Efeito sonoro
efeito = pygame.mixer.Sound("efeito.wav")
efeito.play()
```

AGORA, MEUS AMIGOS, VAMOS CRIAR O NOSSO PRIMEIRO JOGO

15.3 Criando nosso primeiro jogo - Dodge the Blocks

Vamos criar o nosso clássico “Dodge the Blocks”. Neste jogo, o jogador controla um bloco vermelho que deve desviar de blocos azuis caindo do topo da tela. O jogo termina quando o jogador colide com qualquer bloco inimigo. Para facilitar o entendimento, vamos dividir o jogo em partes menores. Cada parte será explicada com o código correspondente e com exemplos claros.

15.3.1 Configuração Inicial

Antes de começar a criar o jogo, precisamos configurar o Pygame e definir os elementos básicos, como a janela do jogo, as cores, e o relógio para controlar a velocidade do jogo.

```
import pygame
import random
import sys

# Inicializar o Pygame
pygame.init()

# Configurações da janela
largura, altura = 800, 600
tela = pygame.display.set_mode((largura, altura))
pygame.display.set_caption("Dodge the Blocks")

# Relógio para controlar o FPS (Frames por Segundo)
relógio = pygame.time.Clock()

# Definir cores no padrão RGB em intensidade - de 0 a 255
BRANCO = (255, 255, 255)
VERMELHO = (255, 0, 0)
AZUL = (0, 0, 255)
```

O que está acontecendo?

- **pygame.init()**: Inicializa todos os módulos do Pygame.

- **Janela do jogo:** Criamos uma janela de 800x600 pixels com `pygame.display.set_mode()`.
 - **Cores:** Utilizamos tuplas para definir cores no formato RGB (Red, Green, Blue).
-

15.3.2 Criando o Jogador

O jogador será representado como um retângulo vermelho. Precisamos definir a posição inicial do jogador, seu tamanho e a velocidade com que ele se move.

```
# Variáveis do jogador
player_x, player_y = 400, 500 # Posição inicial do jogador
player_tamanho = 50           # Tamanho do jogador
player_velocidade = 10        # Velocidade de movimento
```

O que está acontecendo?

- **player_x e player_y:** Coordenadas iniciais do jogador. Ele começa próximo à parte inferior da tela.
 - **player_tamanho:** Define o tamanho do jogador como um quadrado de 50x50 pixels.
 - **player_velocidade:** Determina quantos pixels o jogador se move a cada atualização.
-

15.3.3 Criando os Blocos Inimigos

Os blocos inimigos serão representados como retângulos azuis que começam no topo da tela e se movem para baixo. Precisamos definir a posição, tamanho e velocidade dos blocos.

```
# Variáveis dos blocos inimigos
bloco_largura, bloco_altura = 50, 50 # Tamanho dos blocos
bloco_x = random.randint(0, largura - bloco_largura)

# Posição inicial aleatória no eixo x
bloco_y = -50 # Começa fora da tela, no topo
bloco_velocidade = 7 # Velocidade de movimento
```

O que está acontecendo?

- **bloco_x**: A posição horizontal do bloco é aleatória dentro dos limites da tela.
 - **bloco_y**: O bloco começa fora da tela, na parte superior, e desce com o tempo.
 - **bloco_velocidade**: Define a velocidade de descida dos blocos.
-

15.3.4 Movimentando o Jogador

Precisamos capturar as teclas pressionadas para mover o jogador. O jogador se move para a esquerda ou direita com as setas do teclado.

```
for evento in pygame.event.get():
    if evento.type == pygame.QUIT:
        pygame.quit()
        sys.exit()

# Detectar teclas pressionadas
teclas = pygame.key.get_pressed()
if teclas[pygame.K_LEFT] and player_x > 0:
    # Movimento para a esquerda
    player_x -= player_velocidade
if teclas[pygame.K_RIGHT] and player_x < largura -
    player_tamanho:
    # Movimento para a direita
    player_x += player_velocidade
```

O que está acontecendo?

- **pygame.key.get_pressed()**: Retorna o estado de todas as teclas do teclado.
 - **Condições de movimento**: Garantimos que o jogador não saia dos limites da tela verificando as posições mínima e máxima.
-

15.3.5 Movimentando os Blocos

Os blocos inimigos descem continuamente na tela. Quando saem da parte inferior, reaparecem no topo em uma nova posição horizontal aleatória.

```
# Atualizar posição do bloco
bloco_y += bloco_velocidade
if bloco_y > altura:
    bloco_y = -50 # Reinicia no topo
    bloco_x = random.randint(0, largura - bloco_largura)
    # Nova posição aleatória
```

O que está acontecendo?

- **bloco_y += bloco_velocidade:** Move o bloco para baixo a cada frame.
 - **Reaparecimento:** Quando o bloco atinge o final da tela, ele é reposicionado no topo com uma nova posição horizontal.
-

15.3.6 Detectando Colisões

Precisamos verificar se o jogador colidiu com qualquer bloco inimigo. Isso é feito comparando as posições do jogador e do bloco.

```
# Detectar colisão
if (player_x < bloco_x + bloco_largura and
    player_x + player_tamanho > bloco_x and
    player_y < bloco_y + bloco_altura and
    player_y + player_tamanho > bloco_y):
    print("Game Over!")
    pygame.quit()
    sys.exit()
```

O que está acontecendo?

- **Lógica de colisão:** Verificamos se os retângulos do jogador e do bloco se sobrepõem. Se isso acontecer, o jogo termina.

15.3.7 Desenhando os Elementos

A cada frame, desenhamos o jogador e os blocos na tela. Também limpamos a tela com uma cor de fundo para evitar rastros dos elementos anteriores.

```
# Desenhar elementos na tela

tela.fill(BRANCO) # Cor de fundo

pygame.draw.rect(tela, VERMELHO, (player_x, player_y,
player_tamanho, player_tamanho)) # Jogador

pygame.draw.rect(tela, AZUL, (bloco_x, bloco_y, bloco_largura,
bloco_altura)) # Bloco inimigo

pygame.display.flip() # Atualizar a tela
```

O que está acontecendo?

- **tela.fill()**: Preenche a tela com a cor de fundo (branco).
- **pygame.draw.rect()**: Desenha os retângulos do jogador e dos blocos inimigos.
- **pygame.display.flip()**: Atualiza a tela com as alterações feitas.

15.3.8 Controlando a Velocidade do Jogo

Para evitar que o jogo rode rápido demais, limitamos a quantidade de frames por segundo (FPS).

```
relogio.tick(30) # Limita o jogo a 30 frames por segundo
```

O que está acontecendo?

- **relogio.tick(30)**: Garante que o jogo atualize no máximo 30 vezes por segundo, mantendo a velocidade consistente.

Resumo Final do Fluxo do Jogo

1. **Inicializamos o Pygame** e configuramos os elementos básicos (janela, cores, variáveis).
2. Criamos o **jogador** e controlamos seu movimento com as setas do teclado.
3. Criamos os **blocos inimigos**, que descem continuamente e reaparecem no topo.
4. Verificamos **colisões** entre o jogador e os blocos.
5. Desenhamos todos os elementos na tela e atualizamos a tela em cada frame.
6. Controlamos o FPS para manter a jogabilidade fluida.

Agora, estamos prontos para testar nosso novo jogo! O código completo está no nosso [GitHub](#). Divirtam-se!

15.4 Revivendo clássicos - Snake

Esse aqui todos conhecem, e eu imagino que alguns de vocês já imaginavam que iríamos desenvolver esse projeto aqui na nossa apostila. Do Nokia tijolão para a tela do seu PC: Snake, o famoso ‘jogo da cobrinha’! Vamos estruturar o aprendizado da mesma forma do nosso primeiro jogo.

15.4.1 Configuração Inicial

Antes de começarmos, precisamos configurar o Pygame, a janela do jogo e as cores. Além disso, definimos o tamanho dos blocos que compõem a cobra e a fruta, bem como a velocidade da cobra.

```
import pygame
import random
import sys

# Inicializar o Pygame
pygame.init()

# Configurações da janela
largura, altura = 800, 600
tela = pygame.display.set_mode((largura, altura))
pygame.display.set_caption("Jogo da Cobrinha")
relógio = pygame.time.Clock()
```

```
# Definir cores
BRANCO = (255, 255, 255)
VERDE = (0, 255, 0)
VERMELHO = (255, 0, 0)
PRETO = (0, 0, 0)

# Configurações do jogo
tamanho_bloco = 20 # Tamanho dos blocos da cobra e da fruta
velocidade = 15 # Velocidade do jogo (FPS)
```

O que está acontecendo?

- **Tamanho do bloco (`tamanho_bloco`):** Define o tamanho dos blocos que compõem a cobra e a fruta. Aqui, usamos 20x20 pixels para facilitar os cálculos de movimento.
 - **Velocidade (`velocidade`):** Determina o número de atualizações por segundo, controlando o ritmo do jogo.
-

15.4.2 Criando a Cobra

A cobra é representada por uma lista chamada `corpo_cobra`. Cada elemento dessa lista é um bloco (retângulo) que forma o corpo da cobra.

```
# Variáveis iniciais da cobra
pos_x = largura // 2 # Posição inicial no centro da tela
pos_y = altura // 2
movimento_x = 0 # Movimento inicial
movimento_y = 0
corpo_cobra = [] # Lista que representa o corpo da cobra
comprimento_cobra = 1 # Começa com apenas 1 bloco
```

O que está acontecendo?

- **`pos_x` e `pos_y`:** Representam a posição inicial da cabeça da cobra (no centro da tela).
- **`movimento_x` e `movimento_y`:** Controlam o movimento horizontal e vertical da cobra. Inicialmente, a cobra está parada.

- **corpo_cobra:** É uma lista que armazenará as posições de todos os blocos que compõem a cobra.
-

15.4.3 Criando a Fruta

A fruta é representada como um pequeno retângulo vermelho posicionado em uma coordenada aleatória dentro da tela.

```
# Posição inicial da fruta

fruta_x = random.randint(0, (largura // tamanho_bloco) - 1) *
tamanho_bloco

fruta_y = random.randint(0, (altura // tamanho_bloco) - 1) *
tamanho_bloco
```

O que está acontecendo?

- **Posição aleatória:** A posição da fruta (**fruta_x** e **fruta_y**) é gerada aleatoriamente em múltiplos do tamanho do bloco (**tamanho_bloco**), garantindo que ela fique alinhada com a grade do jogo.
-

15.4.4 Movimentando a Cobra

A cobra se move continuamente em uma direção. O jogador altera a direção com as setas do teclado.

```
for evento in pygame.event.get():
    if evento.type == pygame.QUIT:
        pygame.quit()
        sys.exit()

# Detectar teclas pressionadas

teclas = pygame.key.get_pressed()
if teclas[pygame.K_UP] and movimento_y == 0: # Movimento para
cima

    movimento_x = 0
```

```

    movimento_y = -tamanho_bloco

if teclas[pygame.K_DOWN] and movimento_y == 0: # Movimento
para baixo

    movimento_x = 0
    movimento_y = tamanho_bloco

if teclas[pygame.K_LEFT] and movimento_x == 0: # Movimento
para a esquerda

    movimento_x = -tamanho_bloco
    movimento_y = 0

if teclas[pygame.K_RIGHT] and movimento_x == 0: # Movimento
para a direita

    movimento_x = tamanho_bloco
    movimento_y = 0

# Atualizar posição da cobra
pos_x += movimento_x
pos_y += movimento_y

```

O que está acontecendo?

- **Movimento contínuo:** Após o jogador pressionar uma tecla, a cobra continua se movendo na mesma direção até que outra tecla seja pressionada.
- **Restrição de direção:** A cobra não pode mudar diretamente para a direção oposta para evitar "dobrar" sobre si mesma.

15.4.5 Detectando Colisões

A colisão ocorre em dois casos:

1. **Com as bordas da tela**
2. **Com o próprio corpo da cobra**

```

# Colisão com as bordas
if pos_x < 0 or pos_x >= largura or pos_y < 0 or pos_y >=
altura:

    print("Game Over! Você bateu na parede.")
    pygame.quit()

```

```

sys.exit()

# Colisão com o próprio corpo
for bloco in corpo_cobra[:-1]: # Ignorar a cabeça da cobra
    if bloco == [pos_x, pos_y]:
        print("Game Over! Você bateu no próprio corpo.")
        pygame.quit()
        sys.exit()

```

O que está acontecendo?

- **Bordas:** Se a posição da cabeça da cobra (**pos_x**, **pos_y**) estiver fora dos limites da tela, o jogo termina.
- **Próprio corpo:** Verificamos se a posição da cabeça da cobra coincide com qualquer outro bloco do corpo.

15.4.6 Comendo a Fruta

Quando a cobra come a fruta, seu comprimento aumenta e a fruta reaparece em uma nova posição aleatória.

```

# Detectar se a cobra comeu a fruta

if pos_x == fruta_x and pos_y == fruta_y:

    fruta_x = random.randint(0, (largura // tamanho_bloco) -
1) * tamanho_bloco

    fruta_y = random.randint(0, (altura // tamanho_bloco) - 1)
* tamanho_bloco

    comprimento_cobra += 1

```

O que está acontecendo?

- **Fruta comida:** Se as coordenadas da cabeça da cobra coincidem com as da fruta, o comprimento da cobra é incrementado e a fruta reaparece em uma nova posição.

15.4.7 Atualizando o Corpo da Cobra

A cada frame, atualizamos a posição dos blocos do corpo da cobra.

```
# Atualizar o corpo da cobra
bloco_cabeca = [pos_x, pos_y]
corpo_cobra.append(bloco_cabeca)
if len(corpo_cobra) > comprimento_cobra:
    del corpo_cobra[0] # Remove o bloco mais antigo
```

O que está acontecendo?

- **Cabeça da cobra:** A posição atual da cabeça é adicionada à lista `corpo_cobra`.
 - **Remoção de blocos antigos:** Quando o comprimento da cobra é maior que o necessário, removemos o bloco mais antigo para simular o movimento.
-

15.4.8 Desenhando os Elementos

A cada frame, limpamos a tela e redesenhamos a cobra e a fruta.

```
# Desenhar na tela

tela.fill(PRETO)

pygame.draw.rect(tela, VERMELHO, [fruta_x, fruta_y,
tamanho_bloco, tamanho_bloco]) # Fruta

for bloco in corpo_cobra:

    pygame.draw.rect(tela, VERDE, [bloco[0], bloco[1],
tamanho_bloco, tamanho_bloco]) # Corpo da cobra

pygame.display.flip()
```

O que está acontecendo?

- `tela.fill()`: Preenche a tela com a cor de fundo (preto).
- `pygame.draw.rect()`: Desenha os blocos da cobra e a fruta na tela.
- `pygame.display.flip()`: Atualiza a tela com as alterações feitas.

15.4.9 Controlando a Velocidade

Controlamos a velocidade do jogo ajustando o número de frames por segundo (FPS).

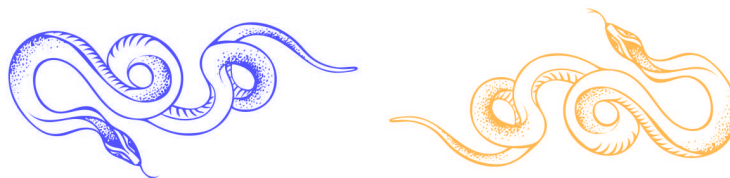
```
relogio.tick(velocidade)
```

O que está acontecendo?

- **relogio.tick(velocidade)**: Limita o número de atualizações por segundo, garantindo que o jogo tenha um ritmo consistente.

Pronto, galera! Esse vai trazer nostalgia pra todos nós! Agora, divirtam-se bastante e, se quiserem, continuem melhorando e otimizando os jogos. Assim como o anterior, o código completo deste jogo estará, também, no nosso [GitHub](#).

A estrutura básica de um jogo envolve inicialização, captura de eventos, atualização e renderização, e Pygame é uma biblioteca poderosa e flexível para criar jogos em 2D! Ele fornece suporte integrado para gráficos, áudio e interação com o usuário. Com o Pygame, você pode explorar sua criatividade e criar desde jogos simples até projetos mais complexos!



Considerações finais

Bem amigos, infelizmente chegamos ao fim. Ao longo desta apostila, exploramos conceitos fundamentais e avançados em programação, cobrindo desde estruturas básicas até áreas complexas como ciência de dados e desenvolvimento de jogos. Cada capítulo foi preparado com muito carinho e dedicação e pensado para oferecer uma base sólida, promovendo o aprendizado prático e o desenvolvimento da lógica de programação.

A programação é uma jornada contínua de aprendizado. Errar faz parte do processo, e cada erro é uma oportunidade de crescimento. Seja criando um jogo no Pygame, treinando um modelo de machine learning ou automatizando tarefas, a prática é o caminho para a excelência.

Lembre-se: a tecnologia está sempre evoluindo, e o conhecimento que você adquiriu aqui é apenas o começo. Continue explorando, experimentando e se desafiando. Não tenha medo de buscar novas ferramentas, projetos e desafios para expandir seus horizontes.

Agradecemos a oportunidade, a confiança e a paciência de todos. Nós desejamos todo o sucesso e felicidade a todos vocês!

Bons estudos e sucesso na sua jornada como programador!