CITHA | Inovação e Sustentabilidade na Amazônia

2025

ALGORITMOS E PROGRAMAÇÃO NA LINGUAGEM PYTHON

(do básico à análise de dados com aplicações práticas)

FOTO DE NEMUEL SERETI

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Algoritmos e programação na linguagem Python [livro eletrônico] : (do básico à análise de dados com aplicações práticas) / Raimundo Fagner Costa...[et al.] ; coordenação Tiago Francisco Andrade Diocesano...[et al.]. -- 2. ed. -- Manaus, AM : Ed. dos Autores, 2025.
PDF

Outros autores: Eduardo Palhares Júnior, Alexandre Lopes Martiniano, Wenndisson da Silva Souza, Nivaldo Rodrigues e Silva.
Outros coordenadores: Jaidson Brandão da Costa, Elcivan dos Santos Silva, Martinho Correia Barros, Adelino Maia Galvão Filho.
Bibliografia.
ISBN 978-65-01-76116-9

1. Algoritmos 2. Dados - Análise 3. Linguagem de programação (Computadores) 4. Python (Linguagem de programação para computadores) I. Palhares Júnior, Eduardo. II. Martiniano, Alexandre Lopes. III. Souza, Wenndisson da Silva. IV. Silva, Nivaldo Rodrigues e. V. Diocesano, Tiago Francisco Andrade. VI. Costa, Jaidson Brandão da. VII. Silva, Elcivan dos Santos. VIII. Barros, Martinho Correia. IX. Galvão Filho, Adelino Maia.

25-310908.0 CDD-005.133

Índices para catálogo sistemático:

1. Python : Linguagem de programação : Computadores : Processamento de dados 005.133

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

DOI: 10.5281/zenodo.15163262

Expediente do IFAM

Reitor
Jaime Cavalcante Alves

Pró-Reitor de Administração
Fábio Teixeira Lima

Pró-Reitor de Gestão de Pessoas
Leandro Amorim Damasceno

Pró-Reitora de Ensino
Rosângela Santos da Silva

Pró-Reitora de Extensão
Maria Francisca Morais de Lima

Pró-Reitor de Pesquisa, Pós-Graduação e Inovação
Paulo Henrique Rocha Aride

Diretor Geral do Campus Manaus Distrito Industrial
Nivaldo Rodrigues e Silva

Expediente do Projeto CITHA

Gestores
Nivaldo Rodrigues e Silva
Samirames da Silva Fleury
Alyson de Jesus dos Santos
Maria Cassiana Andrade Braga
Adanilton Rabelo de Andrade

Coordenadores
Tiago Francisco Andrade Diocesano
Jaidson Brandão da Costa
Elcivan dos Santos Silva
Martinho Correia Barros
Adelino Maia Galvão Filho

Expediente de Produção

Autores
Raimundo Fagner Costa
Eduardo Palhares Júnior
Alexandre Lopes Martiniano
Wenndisson da Silva Souza
Nivaldo Rodrigues e Silva

Avaliação Pedagógica
Samirames da Silva Fleury

Diagramadores
Eduardo Palhares Jr.
Wenndisson da Silva Souza
Fabio Serra Ribeiro Couto

Revisores de Texto
Alyson de Jesus dos Santos

Algoritmos e Programação na Linguagem Python
(do básico à análise de dados com aplicações práticas)

Autores
Raimundo Fagner Costa
Eduardo Palhares Júnior
Alexandre Lopes Martiniano
Wenndisson da Silva Souza
Nivaldo Rodrigues e Silva

Prefácio por Nivaldo Rodrigues e Silva

Revisão
Alyson de Jesus dos Santos

2ª Edição

Manaus - AM
2025

Prefácio

Este e-book é um guia completo sobre programação de computadores, com foco na linguagem Python. Reconhecida por sua simplicidade e versatilidade, Python é uma das linguagens de programação mais populares do mundo. Sua curva de aprendizado acessível e sua ampla aplicabilidade – que abrange desde automação de tarefas até inteligência artificial – tornam-na indispensável para quem deseja explorar o universo da tecnologia.

Estruturado de forma clara e interativa, este material convida o leitor a iniciar sua jornada na programação, oferecendo uma base sólida para o desenvolvimento de códigos e a resolução de desafios práticos. O módulo 4, em especial, exemplifica essa proposta ao demonstrar como aplicar Python na solução de problemas cotidianos. A organização e análise de dados ganham destaque nesse contexto, com o uso de ferramentas e bibliotecas que auxiliam na tomada de decisões de forma eficiente e fundamentada.

Mais do que um recurso acadêmico, este e-book explora aplicações práticas voltadas para a região amazônica, mostrando como a programação pode contribuir para resolver desafios locais. Por meio da otimização de tempo e recursos, o material evidencia o potencial transformador da tecnologia em cenários de alta complexidade e impacto social.

Ao longo das páginas, os leitores serão incentivados a refletir sobre os conceitos fundamentais da lógica de programação e a experimentar casos práticos que conectam teoria e prática. Este e-book não apenas ensina, mas inspira uma mentalidade inovadora e orientada para resultados, mostrando que a programação é uma ferramenta poderosa para enfrentar os desafios do século XXI e promover soluções sustentáveis e transformadoras.

Projeto de Capacitação e Interiorização em Tecnologias Habilitadoras na Amazônia - CITHA

O projeto CITHA surge com o objetivo de fortalecer a economia da Amazônia por meio do incentivo ao empreendedorismo local e do desenvolvimento sustentável. Sua proposta é capacitar profissionais e impulsionar a criação de startups voltadas para a bioeconomia, além de apoiar cooperativas locais na melhoria de seus processos produtivos. A implementação de tecnologias inovadoras é uma das estratégias centrais do projeto, visando oferecer soluções eficientes que atendam às necessidades regionais, como a otimização dos recursos naturais e a melhoria da infraestrutura local.

Ao longo de sua execução, o projeto se compromete a integrar os diversos stakeholders, como governos, empresas, ONGs e comunidades, por meio da capacitação da mão de obra local. O objetivo é formar um capital intelectual qualificado, capaz de apoiar uma governança eficiente, promover a inovação e assegurar a sustentabilidade. O CITHA dedica-se à criação de processos internos que incentivem o desenvolvimento de novos métodos e tecnologias, adaptáveis às particularidades do território amazônico.

Em síntese, o projeto CITHA visa criar um ciclo de desenvolvimento que não só incentive o empreendedorismo, mas também promova a modernização das estruturas locais, elevando a qualidade de vida das populações da Amazônia. Focado em áreas como bioeconomia, inovação e transferência de tecnologia, o projeto busca estabelecer um ecossistema mais forte e autossustentável, capaz de responder eficientemente às demandas do mercado e da sociedade.

Lista de Expressões para Enriquecimento de Conteúdo

Este material foi cuidadosamente estruturado para apoiar sua jornada de aprendizado. Ao longo dos capítulos, você encontrará diversas chamadas sinalizadas por ícones especiais, que ajudarão a destacar pontos-chave e enriquecer sua compreensão. Durante a diagramação, esses ícones serão inseridos conforme as indicações dos autores, guiando você para diferentes tipos de conteúdo e atividades que potencializam seu estudo.

Fique Alerta!

Destaque para conceitos, expressões e trechos fundamentais que merecem sua atenção especial para a compreensão do conteúdo.

Iniciando o diálogo...

Espaço para reflexão crítica. Aqui você será convidado(a) a problematizar os temas abordados, relacionando-os com sua experiência e buscando conexões relevantes para aprofundar seu aprendizado.

Conhecendo um pouco mais!

Indicação de fontes complementares, como livros, entrevistas, vídeos, aplicativos, links e outros recursos para ampliar seu conhecimento sobre o tema.

Caso Prático

Aplicação direta do conteúdo em exemplos concretos, para facilitar a fixação e demonstrar a utilidade do que foi aprendido.

Copie e Teste!

Trechos de código prontos para serem copiados e executados, para que você possa experimentar, validar e explorar na prática os conceitos estudados.

Saída Esperada

```
Apresenta o resultado esperado após a execução de um
bloco de código. Serve como um gabarito para que você
possa verificar se sua implementação está funcionando
corretamente.
```

Sumário

Capítulo 1

Fundamentos da Programação com Python

Iniciando o diálogo...

Bem-vindo(a), estudante! Neste módulo, você iniciará sua jornada no universo da programação com uma das linguagens mais influentes e versáteis da atualidade: o Python. Nosso objetivo é ir além da sintaxe e construir uma base sólida de lógica de programação, capacitando você a resolver problemas de forma estruturada e eficiente. Ao final, você compreenderá não apenas os comandos, mas o raciocínio por trás de um código eficaz.

1.1 Primeiros Passos

1.1.1 O que é Python? Principais Características

Python é uma linguagem de programação de alto nível, interpretada e de propósito geral, o que significa que pode ser usada para desenvolver uma variedade impressionante de aplicações, desde simples scripts de automação até complexos sistemas de inteligência artificial. Criada por Guido van Rossum e lançada pela primeira vez em 1991, sua principal filosofia de design enfatiza a **legibilidade do código** e a simplicidade da sintaxe, permitindo que desenvolvedores expressem conceitos em menos linhas de código do que seria possível em linguagens como C++ ou Java.

A popularidade do Python não é um acaso. Ela se baseia em características poderosas que facilitam o trabalho de programadores iniciantes e experientes:

- **Interpretada:** Diferente de linguagens compiladas (como C++), onde todo o código-fonte é traduzido para a linguagem de máquina antes da execução, o Python é processado em tempo real por um programa chamado *interpretador*. Isso torna o ciclo de desenvolvimento mais rápido e interativo, pois você pode testar pequenas porções de código imediatamente. (Veja a Figura 1.1).

- **Tipagem Dinâmica e Forte:** A tipagem em Python é *dinâmica*, o que significa que você não precisa declarar o tipo de uma variável (inteiro, texto, etc.) antecipadamente. O interpretador infere o tipo durante a execução. Ao mesmo tempo, a tipagem é *forte*, o que impede operações inválidas entre tipos incompatíveis (como somar um número com

um texto sem uma conversão explícita), garantindo mais segurança e previsibilidade ao código (Lutz, 2013).

- **Ecossistema Robusto:** O verdadeiro poder do Python reside em sua vasta coleção de bibliotecas e frameworks, mantida por uma comunidade global ativa. Para análise de dados, temos `Pandas` e `NumPy`; para inteligência artificial, `TensorFlow` e `PyTorch`; para desenvolvimento web, `Django` e `Flask`. Essa riqueza de ferramentas acelera o desenvolvimento e permite a criação de soluções sofisticadas com esforço reduzido.

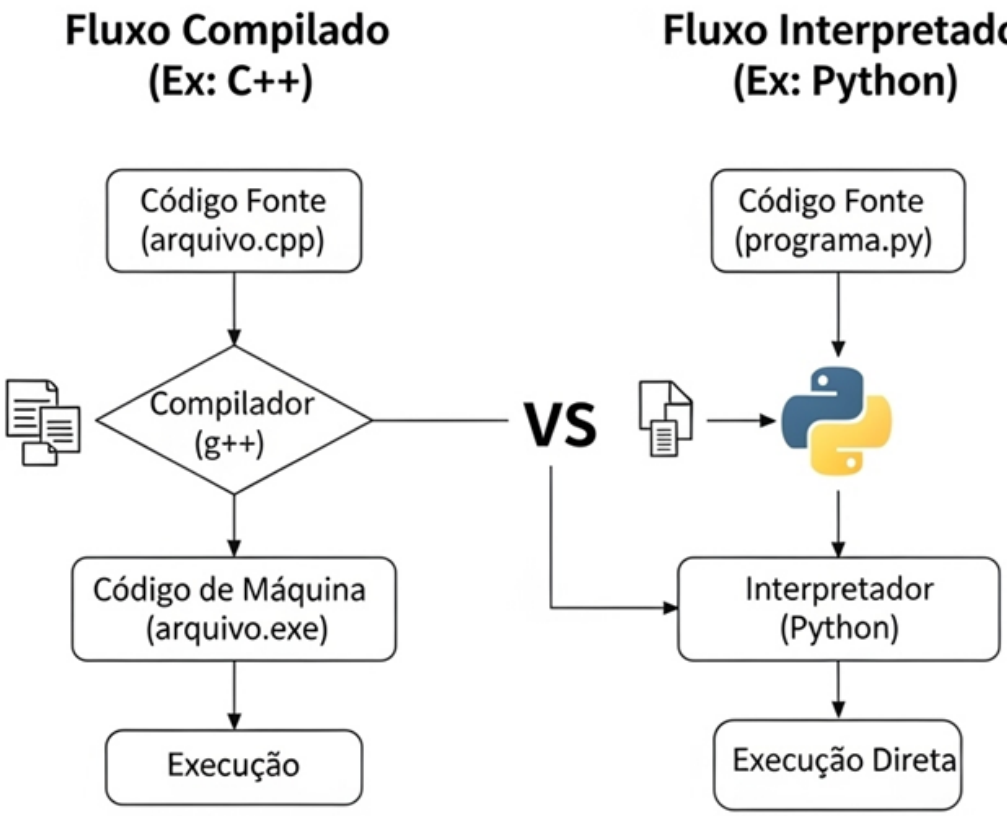

Figura 1.1: Fluxo de uma linguagem compilada vs. uma linguagem interpretada.
Fonte: Gerada por Google Gemini 2.5 Pro, 2025

A simplicidade e o poder do Python fizeram com que ele fosse adotado por gigantes da tecnologia como Google, NASA, Spotify e Netflix, tanto para análise de dados massivos quanto para a infraestrutura de seus serviços.

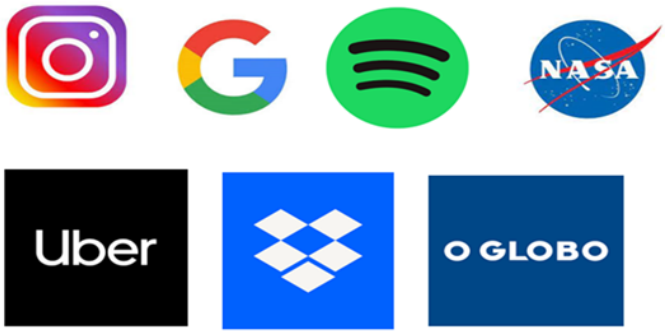

Figura 1.2: Empresas que utilizam Python em seus projetos tecnológicos.
Fonte: Elaborada pelos autores

Conhecendo um pouco mais!

Material complementar
Acesse e conheça um pouco mais sobre a linguagem Python

`https://pt.wikibooks.org/wiki/Python`

ou aponte a câmera do seu smartphone para o Qr Code ao lado

1.1.2 Instalando e Configurando o Ambiente

Agora vamos seguir um guia para instalação do Python no sistema operacional Windows, através dos seguintes passos:

1. Instalação do Python
2. Teste de funcionamento do Python
3. Instalação da IDE
4. Criação de um projeto

Caso queira instalar no linux basta acompanhar o passo a passo em `https://python.org.br/instalacao-linux`.

Instalando o Python

Para realizar a instalação do python, basta acessar o link no site do Python na seção de downloads neste `https://www.python.org/downloads/` e fazer o Download do mesmo conforme apresentado na imagem.

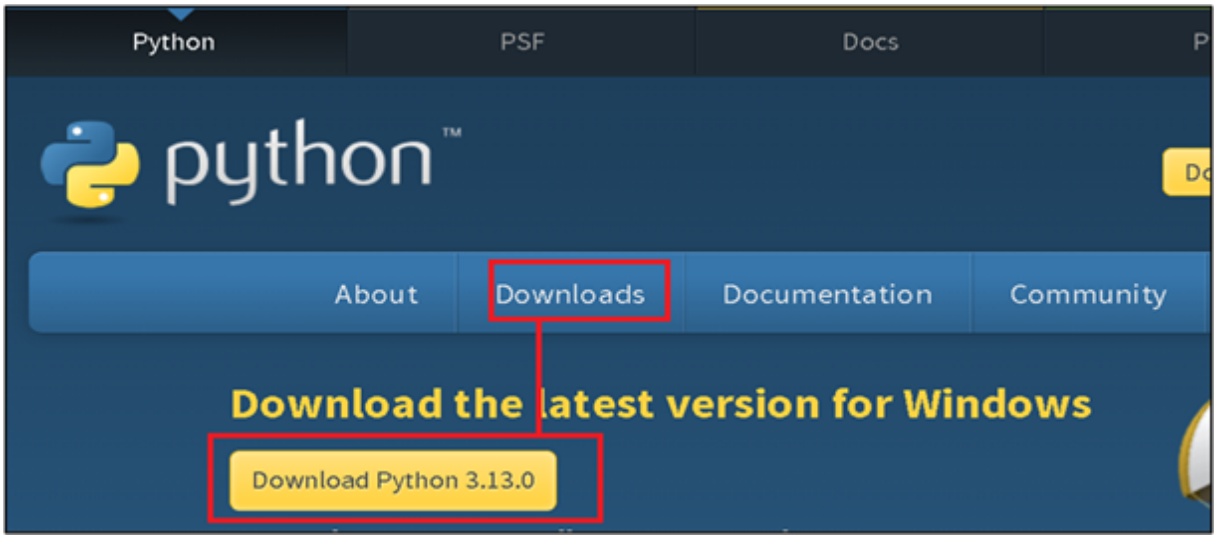

Figura 1.3: Localização do link de download no site oficial

Passo 1: Execute o arquivo baixado e marque as opções destacadas na imagem.

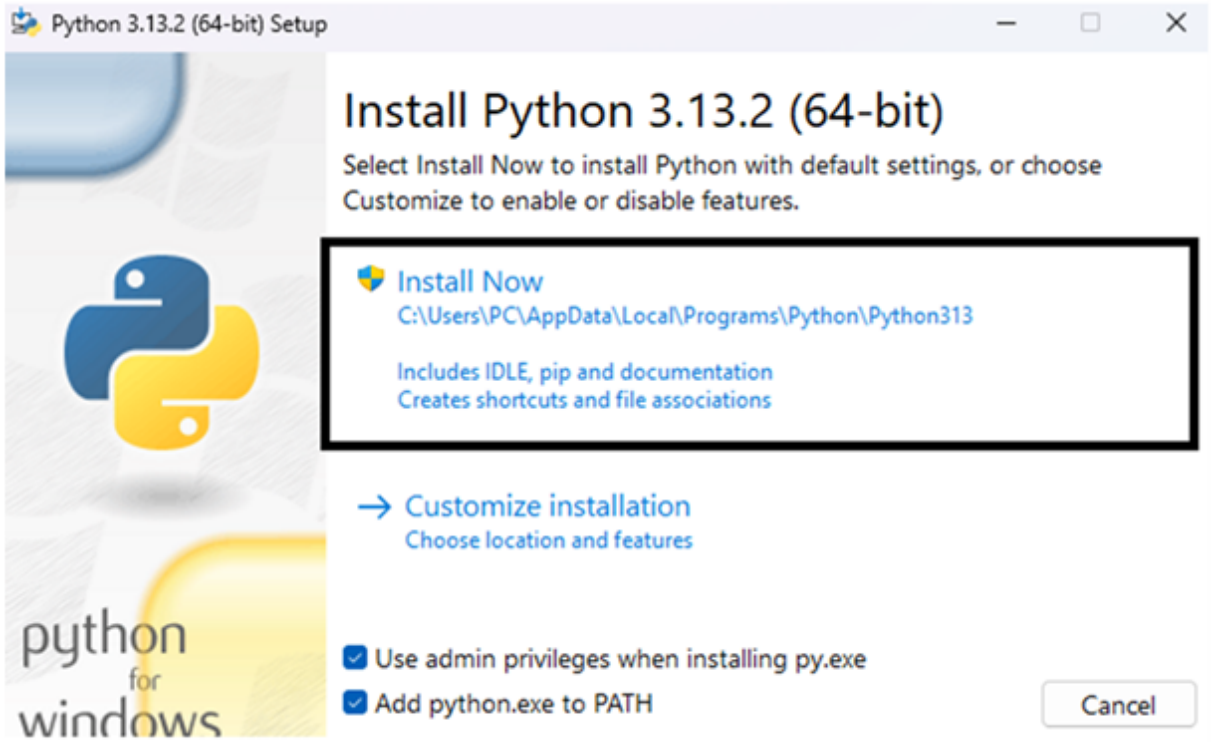

Figura 1.4: Configurações adicionais para instalação customizada

Passo 2: Após realizar a instalação, irá aparecer a tela seguinte, nesta tela clique no botão « Close ».

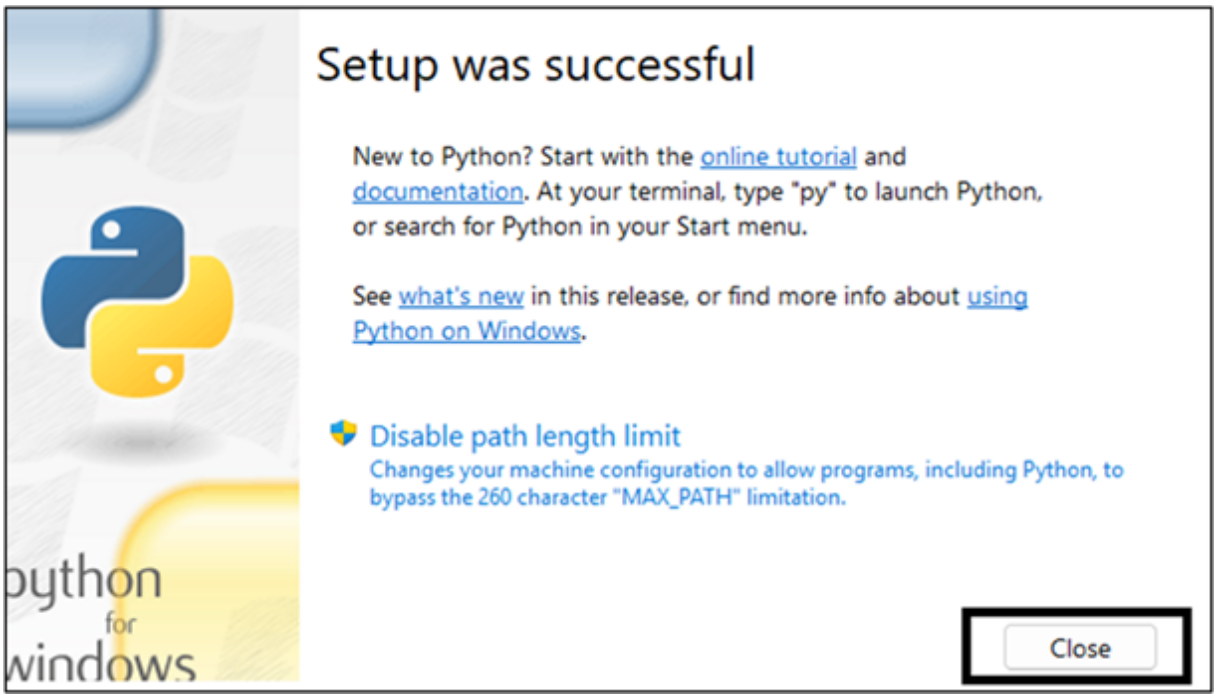

Figura 1.5: Conclusão do processo de instalação

Testando o Python

Após instalar o python, podemos testar se o mesmo está funcionando, ou seja, se foi instalado corretamente. Para isso vá no Menu Iniciar do Windows e digite: CMD e abra o Prompt de Comando.

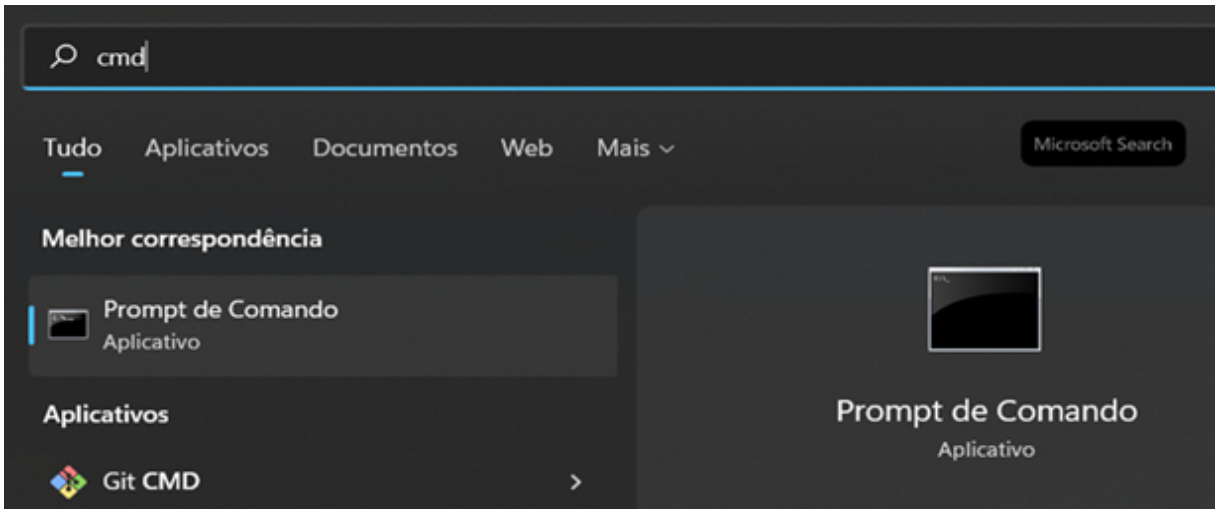

Figura 1.6: Abrindo o prompt de comando - CMD

No Prompt de Comando, ou terminal, digite o comando python, conforme ilustra a imagem abaixo, se aparecer uma mensagem semelhante à apresentada na tela a seguir, é porque deu tudo certo.

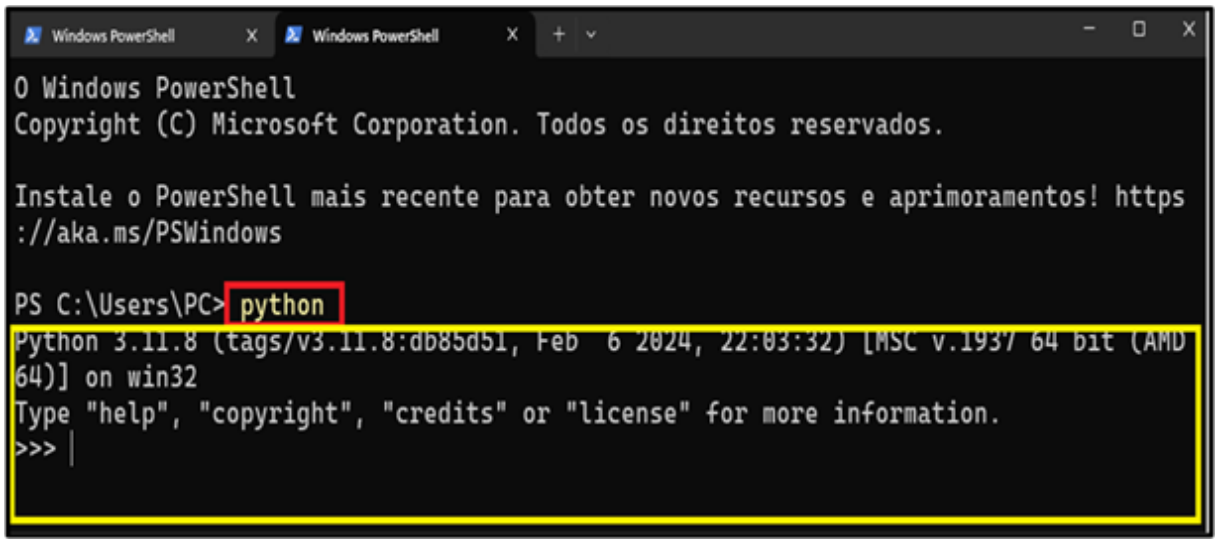

Figura 1.7: Abrindo o prompt de comando - CMD

Instalando o IDE

A forma mais comum para programar em Python é através do uso de um ambiente de programação, também conhecido como ambiente integral de desenvolvimento (Integrated Development Environment – IDE). Mas o que é o IDE? A sigla IDE significa Integrated Development Environment - Ambiente de Desenvolvimento Integrado, em tradução livre é um programa que reúne uma série de ferramentas que facilitam a vida do programador.

Dentre as principais IDEs no mercado para se programar em Python temos:

- Pycharm
- VS Code
- Sublime Text
- Atom
- Jupyter notebook
- Spyder

Neste material usaremos o IDE PyCharm, abaixo seguem as instruções de instalação e como podemos criar um projeto em Python. Para instalar o PyCharm basta acessar o link apresentado na próxima seção.

Acesse e o link abaixo para baixar o editor `https://www.jetbrains.com/pycharm/download/#section=windows`

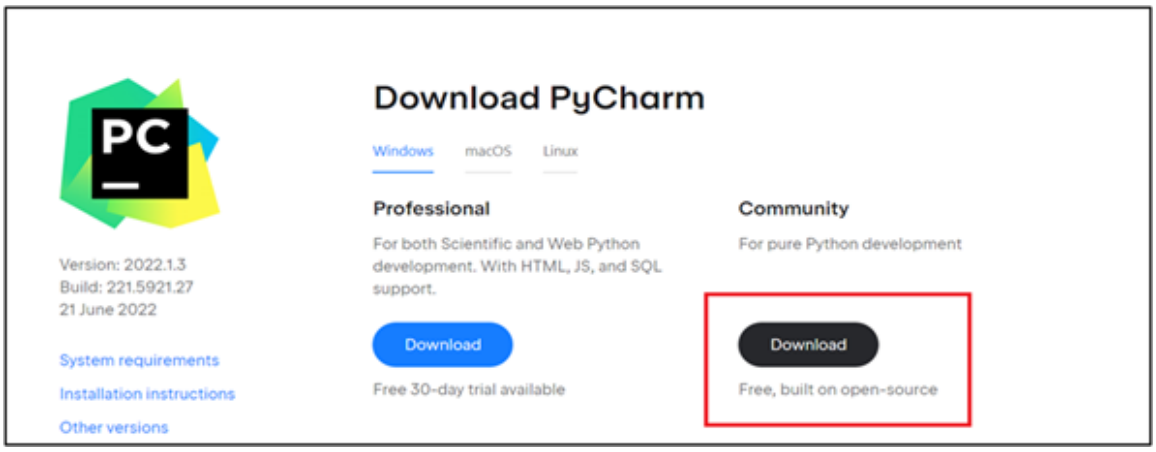

Figura 1.8: Localização do link de download no site oficial

Após realizar o download, realize os passos conforme mostrados a seguir:

Passo 1: Ao executar o arquivo baixado, será mostrado na tela presente na imagem abaixo clique nas opções destacadas na cor vermelho.

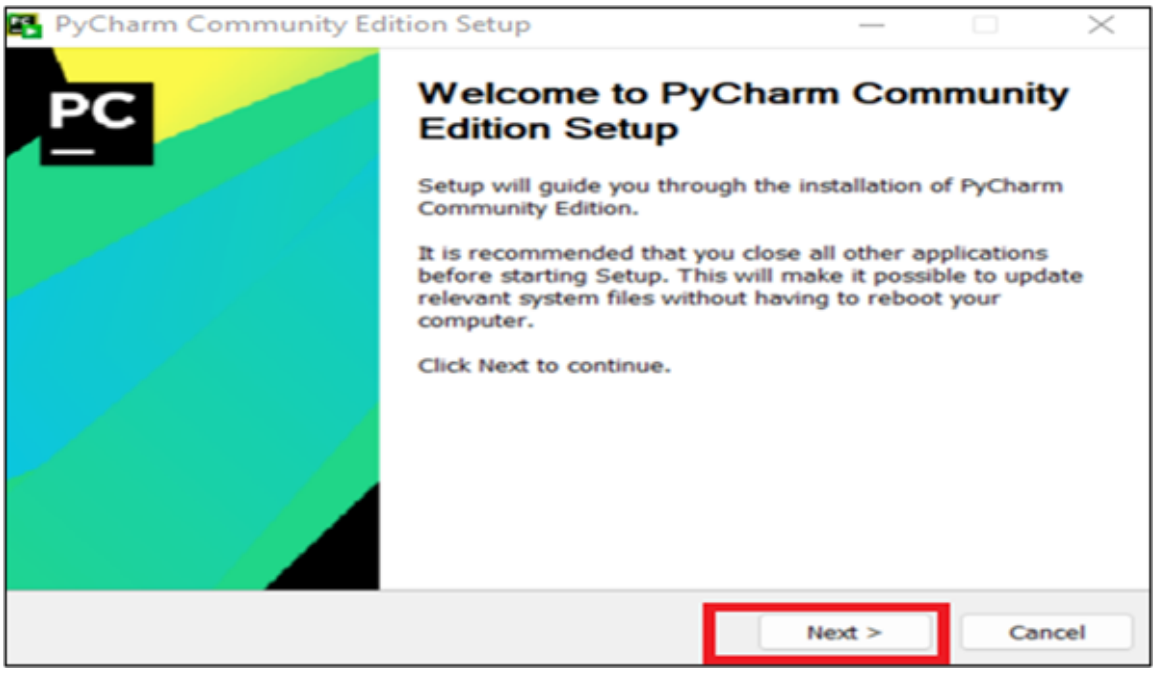

Figura 1.9: Tela inicial do instalador do Pycharm

Passo 2: Na próxima tela deixe as opções padrões. E Clique em «Next »

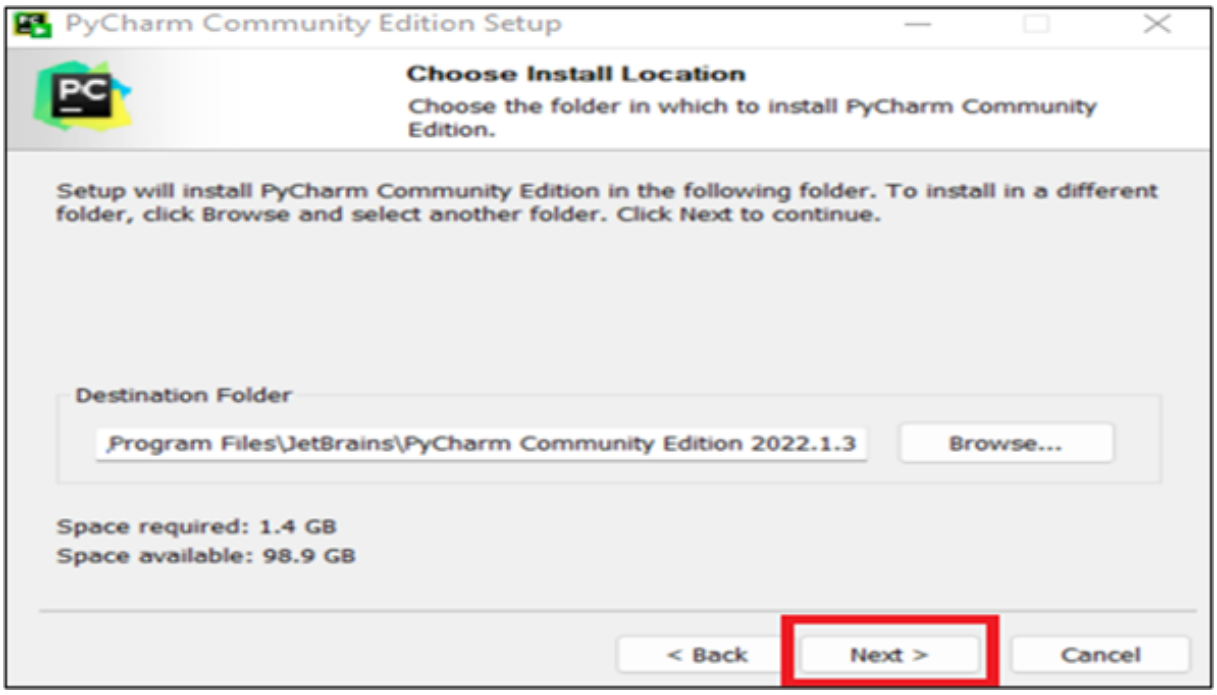

Figura 1.10: Seleção do diretório de instalação do Pycharm

Passo 3: Por fim clique em « Install»

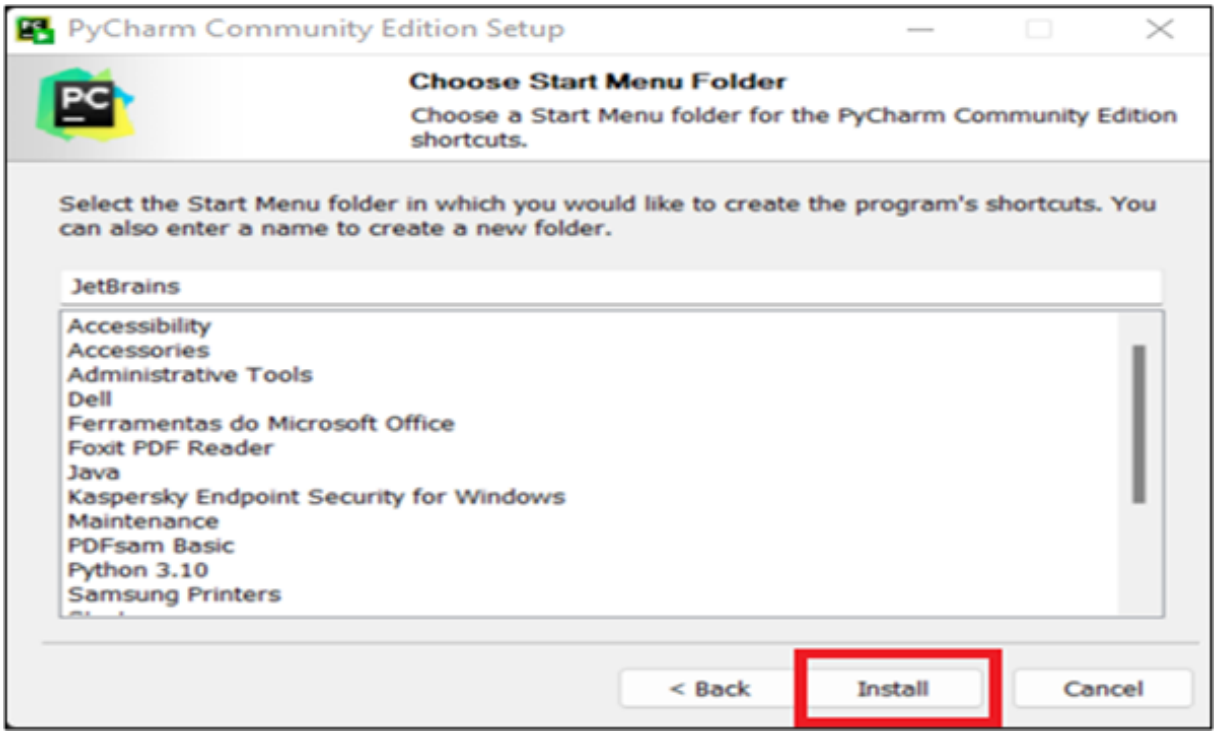

Figura 1.11: Seleção do diretório de atalhos do Pycharm

Criando um projeto no PyCharm

Para criarmos um projeto, abra o programa Pycharm e siga as seguintes etapas:

Passo 1: No Menu File > Acesse opção New Project

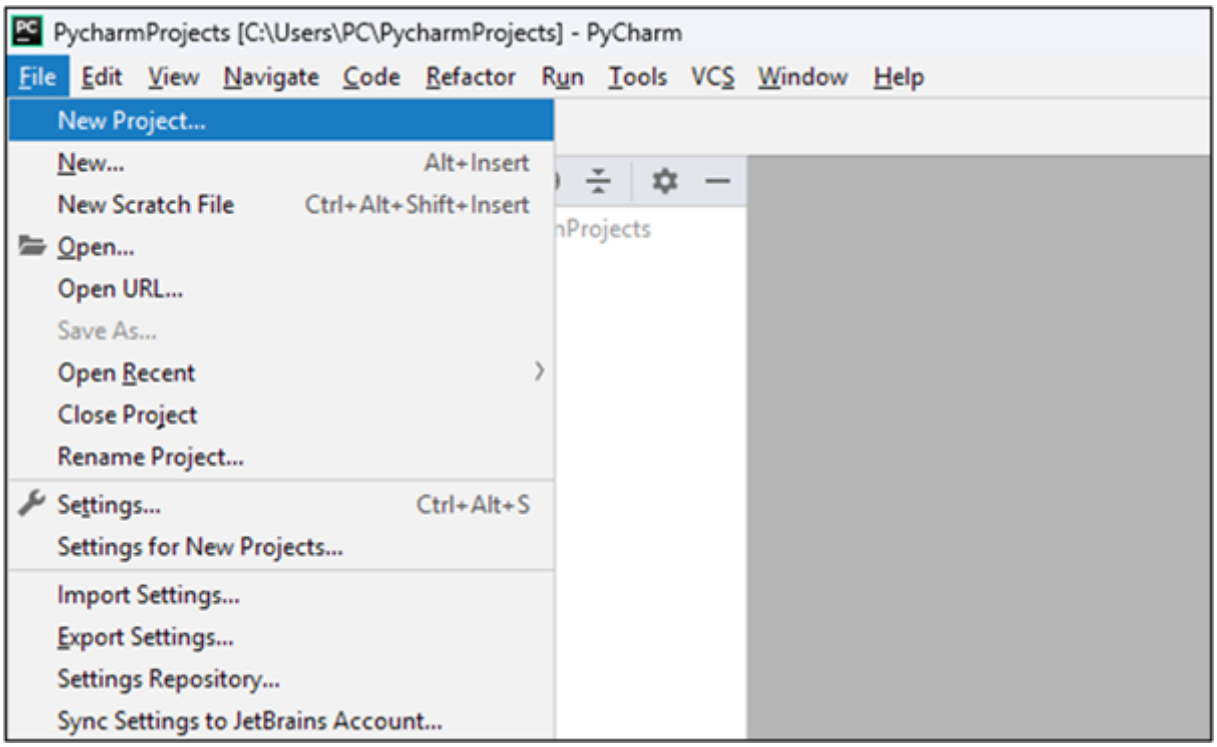

Figura 1.12: Criação de um novo projeto no Pycharm

Passo 2: Defina o local e nome do seu projeto e clique em «Create»

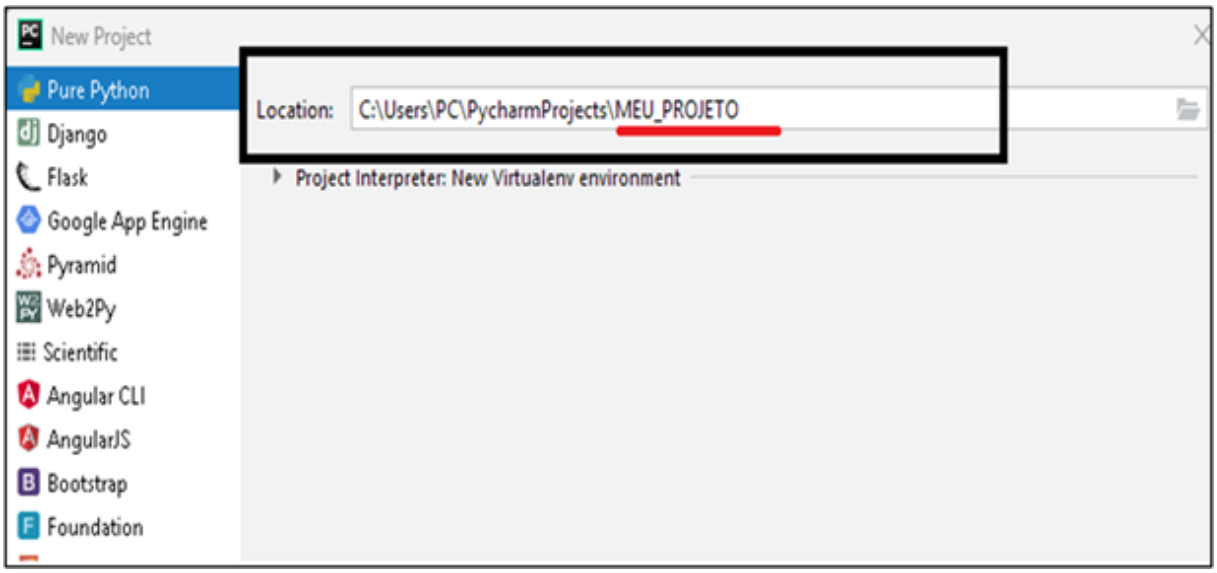

Figura 1.13: Definição do diretório do novo projeto

Passo 3: Após criar o projeto será apresentado a seguinte tela.

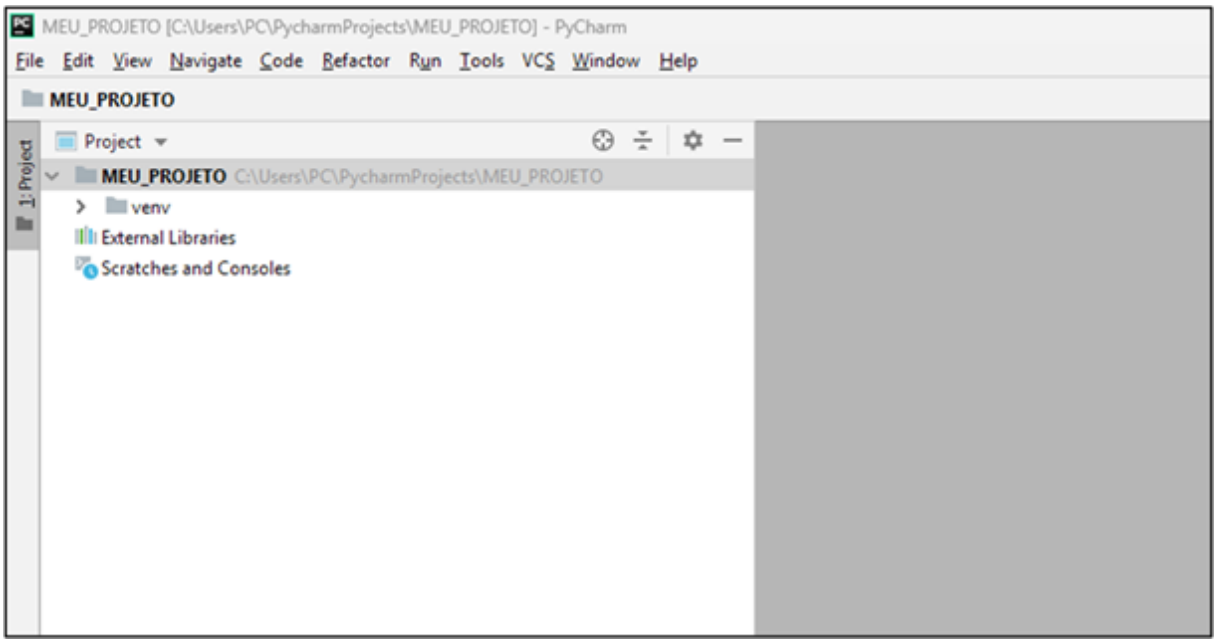

Figura 1.14: Interface do novo projeto em Pycharm

Passo 4: Para criar um arquivo python na pasta do projeto, clique com o botão direito do Mouse na pasta do projeto e vá na opção New Python File, após isso, coloque um nome para o arquivo.

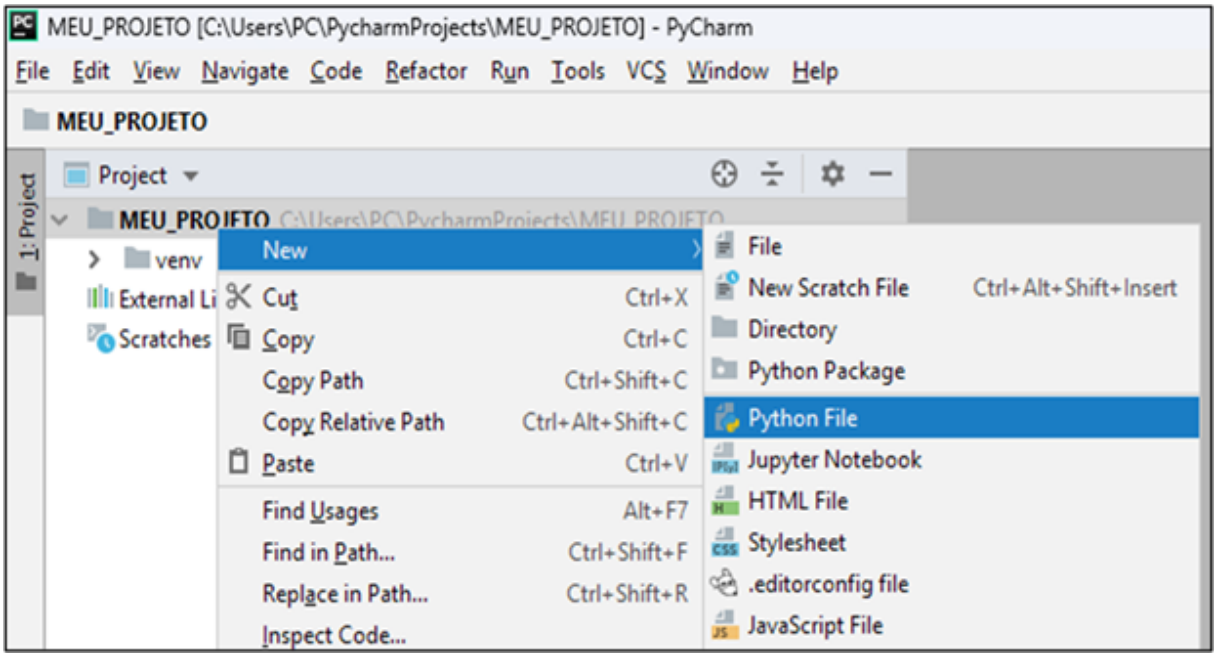

Figura 1.15: Criação de um arquivo Python dentro do novo projeto

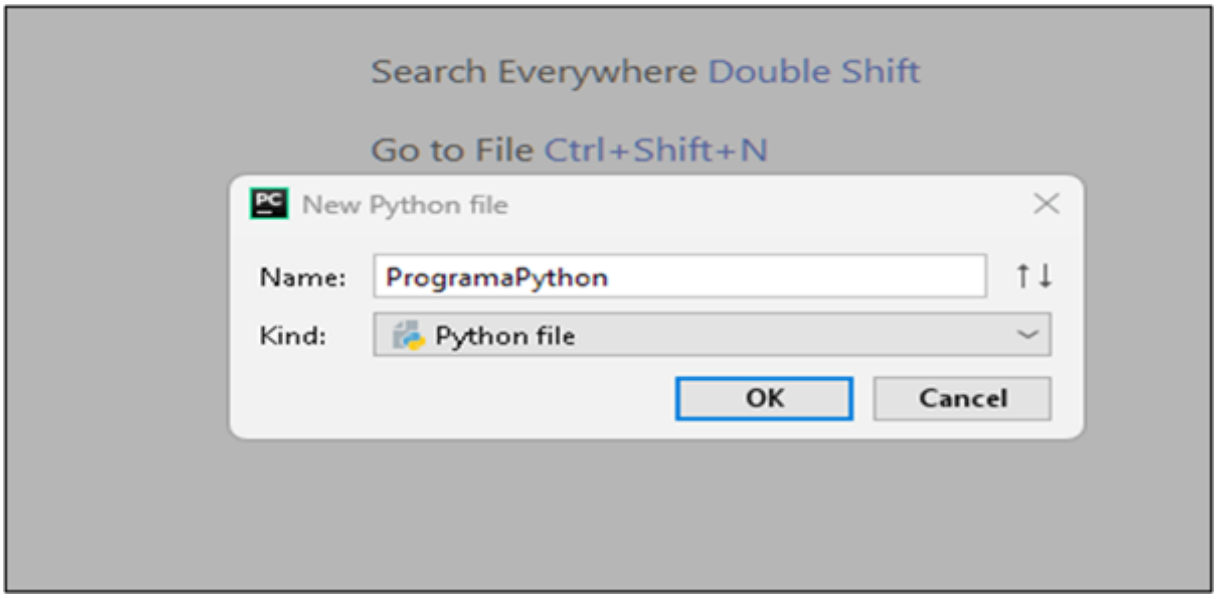

Figura 1.16: Interface para selecionar o nome do arquivo Python

Após isso, seu arquivo será mostrado no IDE, conforme apresentado abaixo.

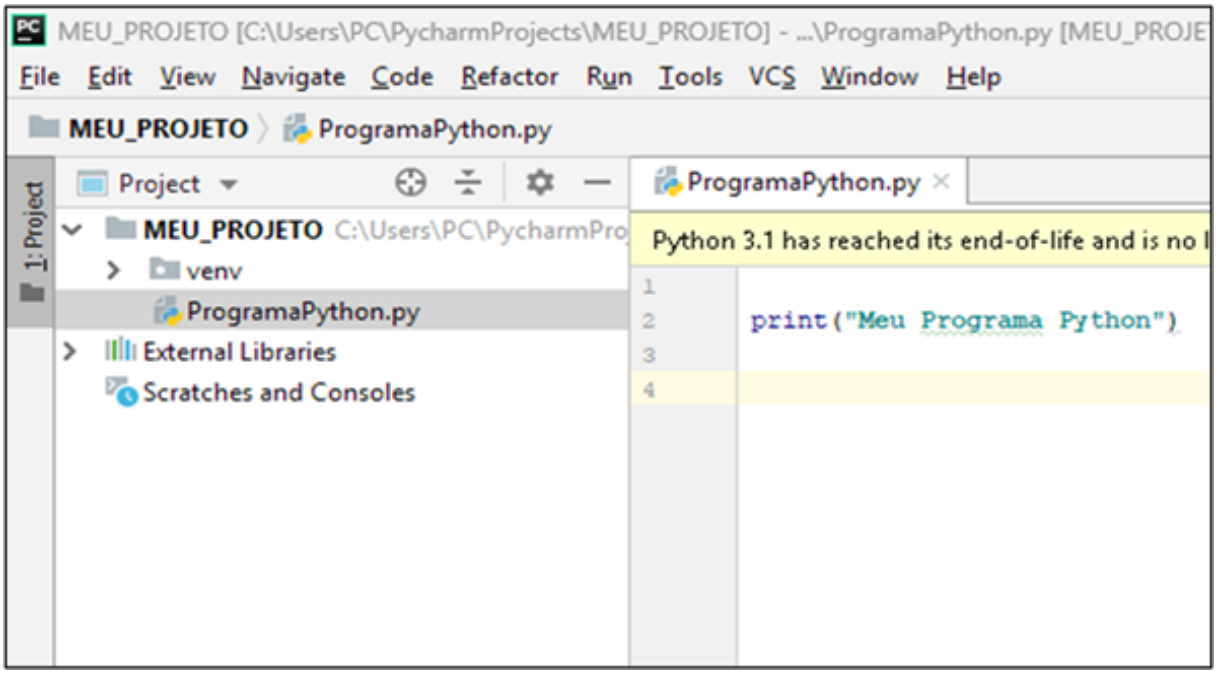

Figura 1.17: Interface do projeto com o arquivo Python

1.1.3 Uma Alternativa na Nuvem: Jupyter Notebooks e Google Colab

Até agora, discutimos a instalação de um ambiente de desenvolvimento tradicional em seu computador. Essa é uma abordagem robusta e amplamente utilizada. No entanto, o mundo da programação e da ciência de dados foi transformado por uma ferramenta que funciona de maneira diferente, mais interativa e visual.

O que é um Jupyter Notebook?

Imagine o caderno de campo de um pesquisador, ele não contém apenas anotações, mas também desenhos, cálculos, amostras coladas e observações. É um documento vivo que conta a história de uma descoberta. O Jupyter Notebook é a versão digital e interativa desse caderno.

Jupyter Notebook é um ambiente de programação que permite criar e compartilhar documentos que contêm, em um só lugar, código, texto e visualizações. Sua estrutura é baseada em dois tipos principais de células:

- **Células de Código:** Contêm código Python que você pode rodar individualmente. O resultado da execução (textos, tabelas, gráficos) aparece imediatamente abaixo da célula.
- **Células de Texto:** Permitem que você escreva explicações, anotações, títulos e links usando uma linguagem simples chamada Markdown para formatar o texto.

Essa estrutura torna o Jupyter ideal para a análise de dados exploratória, pois permite que você execute o código passo a passo, documente seu raciocínio e visualize os resultados em tempo real, contando a história por trás dos dados.

Google Colaboratory (Colab): O Jupyter na Nuvem

E se você pudesse ter acesso a esse caderno de campo digital de qualquer lugar, sem precisar instalar nada, usando apenas seu navegador de internet? Essa é a revolução que o **Google Colab** proporciona.

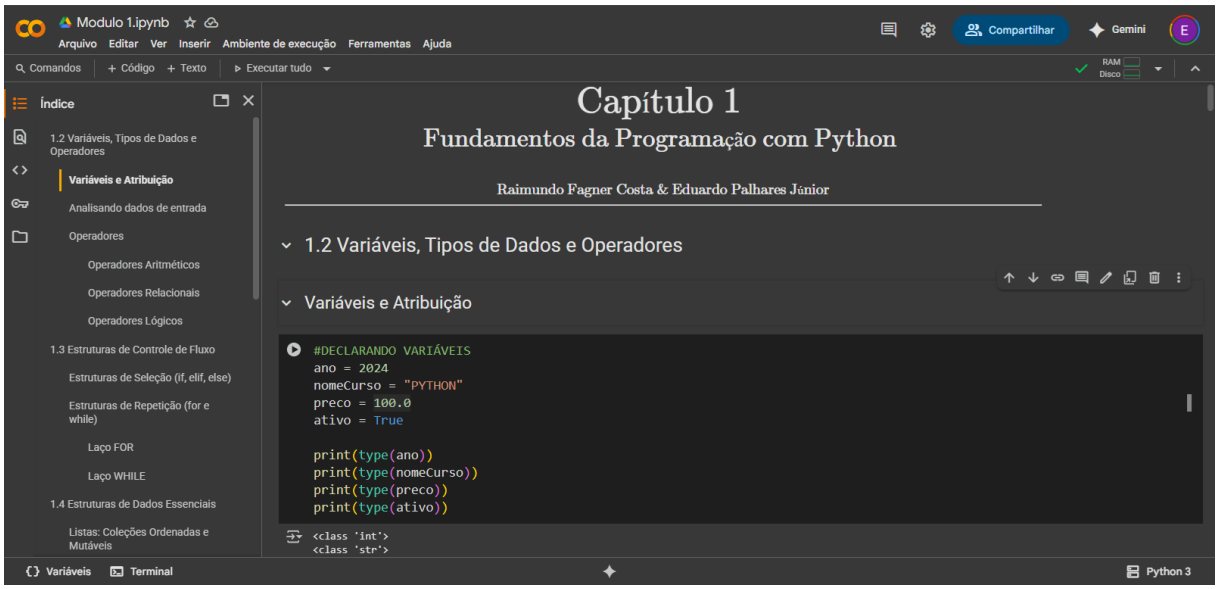

Figura 1.18: Interface do Google Colaboratory

O Colab é, essencialmente, a versão do Jupyter Notebook oferecida gratuitamente pelo Google que roda direto do navegador web. Ele elimina as barreiras de entrada para a programação, tornando-a muito mais acessível e democrática. Suas principais vantagens são:

- **Zero Configuração:** Não é necessário instalar Python e suas bibliotecas básicas. Tudo funciona diretamente no seu navegador, conectado a um computador do Google.
- **Colaboração Fácil:** Você pode compartilhar seus notebooks com outras pessoas, da mesma forma que compartilha um Google Docs, permitindo o trabalho em equipe.

- **Recursos Gratuitos:** O Colab oferece acesso gratuito a recursos computacionais poderosos, como GPUs, que são essenciais para tarefas mais avançadas de Machine Learning e Inteligência Artificial.

Como pode ser observado na figura 1.18, é possível explorar os blocos de texto para criar vizualizações completamente customizadas, incluindo equações LaTeX, figuras e vídeos. Além disso, em projetos muito grandes a estrutura de índices ajuda muito a navegar pelo arquivo e realizar ostestes de forma segmentada.

1.2 Variáveis, Tipos de Dados e Operadores

1.2.1 Variáveis e Atribuição

Na programação de computadores, conforme discutido por Mueller (2020), a entrada, o processamento e a saída de dados podem manipular vários tipos de dados, tais como números letras - caracteres alfanuméricos - e booleanos. Da mesma forma, na linguagem Python existem diferentes tipos de dados, podendo ter dados nos formatos numéricos, textuais e booleanos. Dentro os principais tipos de dados, podemos destacar:

- **int** - Representa os números inteiros: 7, 10, 2022,70
- **float** - Representa os números reais: 3.2, 1.7, 10.4
- **str** - Representa as strings: “Instituto Federal do Amazonas”, “CITHA”, “olá mundo”
- **bool** - Representa os valores booleanos: Verdadeiro (True) ou Falso (False)

No Python, a função `Type` pode ser utilizada para mostrar o tipo de um dado. A função `Type` recebe um argumento e retorna seu tipo mostrado em tela.

Caso Prático

Crie um script PYTHON para testar a declaração variáveis e imprimir os tipos de cada uma.

Copie e Teste!

```
import numpy as np
import matplotlib.pyplot as plt

#DECLARANDO VARIÁVEIS
ano = 2024
nomeCurso = "PYTHON"
preco = 100.0
ativo = True

print(type(ano))
print(type(nomeCurso))
print(type(preco))
print(type(ativo))
```

Saída Esperada

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
```

Ao executar o código, podemos verificar a saída do terminal com o tipo de cada variável, sendo a variável `ano` do tipo int (numero inteiro), a variável `nomeCurso` do tipo str (string ou palavra), e a varável `preco` do tipo float (numero decimal ou ponto flutuante) e a variável `ativo` é bool (booleana ou lógica).

Sempre que executamos um código ou algoritmo, surge a necessidade de armazenarmos dados temporários na memória, e para isso temos o que chamamos de variáveis. Todas as variáveis utilizadas em algoritmos devem ser declaradas antes de serem utilizadas. Isto se faz necessário para permitir que o compilador reserve um espaço na memória para as mesmas.

Segundo Ramalho (2015), variáveis atuam como containers que armazenam dados na memória volátil, para serem acessadas durante a execução de um programa. Como o Python é uma linguagem de tipagem dinâmica (variável) e forte (valor), a declaração de variáveis em Python é implícita. Em Python as variáveis podem ser dos tipos: Texto (String), Numérica (Inteiro ou Real) e lógicas (Booleanas - bool).

Para declarar uma variável é utilizado o operador `= (Igual)` que atribui valores às variáveis. Após ter declarado as variáveis, podemos acessá-las a qualquer momento para fazer alguma operação, como cálculos, comparações ou qualquer outro tipo de manipulação.

Copie e Teste!

```python
#DECLARANDO VARIÁVEIS
ano = 2024
nomeCurso = "PYTHON"
preco = 100.0
ativo = True

print("ANO:", ano)
print("NOME CURSO:", nomeCurso)
print("PREÇO:", preco)
print("ATIVO:", ativo)
```

Saída Esperada

```
ANO:2024
NOME CURSO: PYTHON
PREÇO: 100.0
ATIVO: True
```

Ao se criar novas variáveis, é recomendado seguir algumas regras, também conhecido como conjunto de boas práticas, conforme apresentado abaixo:

- Nomes de variáveis devem começar com letras minúsculas ou sublinhado,

- Não se deve declarar variáveis com caracteres especiais (@, #, $, %, &)
- A linguagem Python é case sensitive, isto é, diferencia letras maiúsculas e minúsculas

1.2.2 Analisando dados de entrada

Ao ler dados do teclado, o Python faz a leitura no formato de Str (texto), sendo assim se quisermos trabalhar com números devemos converter o valor lido para o tipo desejado.

Copie e Teste!

```
#DECLARANDO VARIÁVEIS
idadeTexto = input("Digite sua idade: ")
idadeNumero = int(input("Digite sua idade: "))

print(type(idadeTexto))
print(type(idadeNumero))
```

Saída Esperada

```
Digite sua idade: 100
Digite sua idade: 100
<class 'str'>
<class 'int'>
```

Observe que apesar de praticamente iguais são duas variáveis diferentes com tipos diferentes, já que o comando `int(input("Digite sua idade: "))` converte o texto em um número do tipo float.

Fique Alerta!

A conversão de dados é importante principalmente quando vamos realizar operações matemáticas com números.

Como o Python trata os dados de forma rigorosa, ele não permite, por exemplo, somar um número a um texto. Veja dois cenários práticos onde a conversão é indispensável:

- **Leitura de Entradas do Usuário:** Quando um programa solicita a idade de uma pessoa com a função input(), o Python captura esse dado como uma string (texto). Se o usuário digitar ”25”, o programa armazena o texto ”25”. Tentar realizar uma operação como idade + 1 resultaria em um erro. É necessário converter explicitamente a string ”25”para o inteiro 25 usando a função int() para que o cálculo seja viável.
- **Cálculos Financeiros e Científicos:** Imagine um sistema que calcula o preço final de um produto. A quantidade pode ser um número inteiro (int), mas o preço unitário é um número de ponto flutuante (float). Para garantir a precisão do resultado, pode ser necessário converter o valor da quantidade para float antes da multiplicação, assegurando que o resultado final mantenha as casas decimais corretas, algo essencial em contextos financeiros.”

O uso de comentários é uma prática normal na programação. O objetivo é poder adicionar descrições em partes específicas do código, seja para documentá-lo, seja para adicionar uma descrição, ou mesmo, para marcar uma determinada linha, ou um conjunto de linhas, ao executar o código os comentários não são executados pelo interpretador.

Em Python, podemos fazer comentários de uma linha colocando o carácter # à frente do texto, ou de múltiplas linhas ‘’’ texto ‘’’. Veja os exemplos

Copie e Teste!

```
#DECLARANDO Variáveis
idadeTexto = input("Digite sua idade: ")
idadeNumero = int(input("Digite sua idade: "))

'''
Abaixo é realizada a impressão de cada tipo de variável.
'''

print(type(idadeTexto)) # versão em texto
print(type(idadeNumero)) # versão em número
```

1.2.3 Operadores

Operadores são elementos que nos permitem avaliar dados pelo computador, neste caso por um algoritmo. Por meio expressões que podem ser classificadas como:

- Aritméticos
- Lógicos
- Relacionais

Operadores Aritméticos

Utilizados para realizar cálculos matemáticos.

- **Adição (+)**, **Subtração (-)**, **Multiplicação (*)**, **Divisão (/)**
- **Divisão Inteira (//)**: Realiza a divisão e descarta a parte fracionária. Ex: `10 // 3` resulta em `3`.
- **Módulo (%)**: Retorna o resto de uma divisão. Ex: `10 % 3` resulta em `1`. É muito útil para verificar se um número é par ou ímpar.
- **Exponenciação (**)**: Eleva um número a uma potência. Ex: `2 ** 3` resulta em `8`.

Caso Prático

Imagine que você deve resolver o seguinte problema com a linguagem python: Calcular qual a média de um aluno na disciplina de português,onde foram aplicadas 4 provas. A média do aluno será a soma das 4 notas dividido pela quantidade total de provas.

Copie e Teste!

```
#Declaração de Variáveis
prova1 = 5
prova2 = 10
prova3 = 6
prova4 = 10
quantidadeDeProvas = 4

#Calculando a média
media=(prova1+prova2+prova3+prova4) / quantidadeDeProvas

#Imprimindo na tela a média
print("Media = ", media)
```

```
Media =  7.75
```

Operadores Relacionais

Operadores Relacionais são utilizados para comparar Strings de caracteres (Textos) e números. O resultado da comparação retorna um valor booleano (Lógico) podendo ser verdadeiro ou falso, ou seja, True ou False. Dentre os operadores relacionais, podemos destacas:

Tabela 1.1: Operadores Relacionais em Python.

Operador	Conceito	Exemplo
`>` (Maior que)	Verifica se um valor é maior que outro	`5 > 3`
`<` (Menor que)	Verifica se um valor é menor que outro	`5 < 3`
`==` (Igual a)	Verifica se um valor é igual a outro	`5 == 4`
`>=` (Maior ou igual a)	Verifica se um valor é maior ou igual a outro	`5 >= 5`
`!=` (Diferente de)	Verifica se um valor é diferente de outro	`5 != 5`
`<=` (Menor ou igual a)	Verifica se um valor é menor ou igual a outro	`5 <= 5`

Caso Prático

Imagine que você deve resolver o seguinte problema com a linguagem python: Calcular qual a média de um aluno na disciplina de português,onde foram aplicadas 4 provas. A média do aluno será a soma das 4 notas dividido pela quantidade total de provas.

Copie e Teste!

```
primeiroNumero = 10
segundoNumero = 20

print("Resultado : ", primeiroNumero > segundoNumero)
```

```
Resultado: False
```

Conhecendo um pouco mais!

Material complementar
Conheça um pouco mais sobre a operadores aritméticos em python.

`https://www.youtube.com/watch?v=bfuPfTT5fgI`

ou aponte a câmera do seu smartphone para o Qr Code ao lado

Operadores Lógicos

Em programação, nem sempre uma decisão depende de uma única condição. Muitas vezes, para que um algoritmo tome o caminho correto, ele precisa avaliar múltiplos cenários simultaneamente. É nesse contexto que entram os **operadores lógicos**.

Eles funcionam como conectivos que nos permitem combinar o resultado de duas ou mais expressões, transformando múltiplos resultados em uma única resposta booleana (`True` ou `False`). Os operadores lógicos são as ferramentas que unem essas verificações individuais em uma decisão final e coesa.

Tabela 1.2: Tabela de Operadores Lógicos em Python.

Operador	Conceito	Exemplo
`and`	Retorna `True` se todas as condições avaliadas forem verdadeiras; caso contrário, retorna `False`.	`5 > 3 and 1 == 2`
`or`	Retorna `True` se pelo menos uma das condições for verdadeira. Retorna `False` somente se todas forem falsas.	`5 < 3 or 1 == 2`
`not`	Inverte o resultado booleano: se a expressão for `True`, retorna `False`, e vice-versa.	`not (5 == 4)`

Caso Prático

Imagine que você deve resolver o seguinte problema com a linguagem python: verifique se dois números são maiores que 10, neste caso você vai comportar se os dois números são maiores por meio de um operador relacionais e lógicos.

Copie e Teste!

```
primeiroNumero = 10
segundoNumero = 20

print("Resultado : ", (primeiroNumero > 10) and (segundoNumero >
    10) )
```

```
Resultado: False
```

Conhecendo um pouco mais!

Material complementar
Conheça um pouco mais sobre a operadores relacionais em python.

`https://www.youtube.com/watch?v=ijgle1EJ2XM`

ou aponte a câmera do seu smartphone para o Qr Code ao lado

1.3 Estruturas de Controle de Fluxo

Até agora, vimos como um programa executa comandos de forma linear e sequencial, do início ao fim do arquivo. No entanto, o verdadeiro poder da programação reside na capacidade de desviar desse fluxo, permitindo que o código tome decisões, execute tarefas repetidamente e reaja a diferentes condições. Para isso, utilizamos **estruturas de controle de fluxo**, que são os blocos de construção lógicos que governam a ordem em que as instruções são executadas. Nesta seção, exploraremos os dois tipos fundamentais de controle de fluxo em Python:

- **Estruturas de Seleção** (`if`, `elif`, `else`): Permitem que o programa escolha qual bloco de código executar com base em uma condição ser verdadeira ou falsa.
- **Estruturas de Repetição** (`for`, `while`): Usadas para executar o mesmo bloco de código múltiplas vezes, automatizando tarefas repetitivas.

Antes de mergulharmos nessas estruturas, abordaremos um conceito que é a base para a organização de todas elas em Python: a **indentação**.

1.3.1 Indentação: A Base da Estrutura em Python

A indentação consiste em organizar o código, deslocando algumas linhas para a direita ao inserir espaços no início delas. Embora em muitas linguagens de programação a indentação seja apenas uma questão de estilo, no Python ela é obrigatória.

No Python, a indentação desempenha um papel fundamental, pois além de tornar o código mais legível, é necessária para que ele funcione corretamente. Caso a indentação não esteja adequada, o programa pode apresentar erros ou comportamentos imprevisíveis.

Conhecendo um pouco mais!

Material complementar
veja como funciona e qual a importância de indentação no Python

`https://www.youtube.com/watch?v=ijgle1EJ2XM`

ou aponte a câmera do seu smartphone para o Qr Code ao lado

1.3.2 Estruturas de Seleção (if, elif, else)

Uma estrutura de Seleção ou Condição - permite a escolha de um conjunto de ações ou instruções que serão executadas quando determinadas condições, representadas por expressões lógicas ou relacionais, sejam satisfeitas ou não. Em Python uma estrutura de seleção funciona com a palavra ”if”segundo a estrutura.

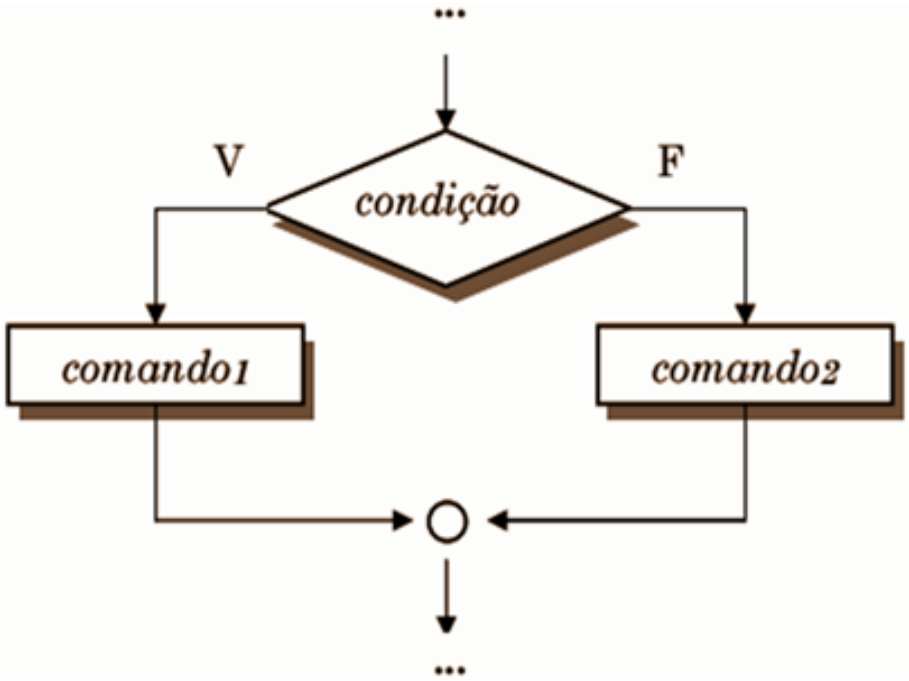

Figura 1.19: Fluxograma que representa o funcionamento da estrutura if/else
Fonte: Mechmaniacs, 2017

Dentre as estruturas de seleção podemos ter condições simples e compostas, como veremos maiores detalhes a seguir. Estrutura de Seleção Simples: É usada quando precisamos testar uma certa condição antes de executar uma ação. Para realizar uma condição podemos usar a palavra (if). As estruturas de seleção são usadas para avaliar expressões relacionais e lógicas tais como:

- Igualdade (a == b)
- Diferentes (a != b)
- Menor que (a < b)
- Menor ou igual (a <= b)
- Maior ou igual (a >= b)
- True == True
- True == False
- (a < b) and (b > c)
- (a < b) or (b > c)

Caso Prático

Imagine que você deve resolver o seguinte problema com a linguagem python: Imagine que você deve fazer um programa em python para saber se uma pessoa é maior de idade, neste caso se a idade da pessoa for maior ou igual a 18, você deve imprimir na tela ”VOCÊ É MAIOR DE IDADE”.

Atenção: observe que se a expressão idade >= 18 for verdadeira, serão realizadas todas as ações que estão “dentro” do bloco, caso a condição não for verdadeira, não será impresso nada em tela.

Copie e Teste!

```
idade = 20

if idade >= 18:
     print("VOCÊ É MAIOR DE IDADE")
```

Saída Esperada

```
VOCE É MAIOR DE IDADE
```

Quando duas alternativas dependem de uma mesma condição utilizamos o IF...ELSE, conforme representado na figura 1.19. O trecho de código dentro do ELSE será executado caso o teste realizado pelo IF não seja verdadeiro.

Caso Prático

Vamos aprimorar nosso programa anterior. Agora, além de informar quando uma pessoa é maior de idade, também queremos uma resposta caso ela seja menor. Utilizando a estrutura if/else, o programa deve checar se for maior ou igual a 18 e imprimir ”VOCÊ É MAIOR DE IDADE”. Caso contrário, ele segue o caminho alternativo e imprimir ”VOCÊ É MENOR DE IDADE”.”

Copie e Teste!

```
idade = 15

if idade >= 18:
   print("VOCÊ É MAIOR DE IDADE")
else:
   print("VOCÊ É MENOR DE IDADE")
```

Saída Esperada

```
VOCE É MENOR DE IDADE
```

Conhecendo um pouco mais!

Material Extra
Conheça um pouco mais sobre estruturas de seleção em Python.

`https://www.youtube.com/watch?v=x74U7k7YoR8`

ou aponte a câmera do seu smartphone para o Qr Code ao lado

1.3.3 Estruturas de Repetição (for e while)

Muitas das tarefas mais poderosas da programação envolvem a automação de processos repetitivos. Imagine ter que processar cada um dos mil itens de um estoque ou calcular a média de notas para cada aluno de uma turma. Escrever o código para cada item individualmente seria impraticável.

É para resolver esse tipo de problema que existem as **estruturas de repetição**, também conhecidas como **laços** ou *loops*. O domínio dessas estruturas é um passo essencial no aprendizado de algoritmos e lógica, sendo um pilar na introdução à programação com Python (Menezes, 2019). Elas nos dão o poder de definir um bloco de código e instruir o computador a executá-lo múltiplas vezes, seja um número fixo de vezes ou enquanto uma determinada condição for verdadeira. Em Python, existem duas principais estruturas para realizar essa tarefa:

- O laço **`for`**, utilizado para iterar sobre os elementos de uma sequência (como uma lista ou uma tupla). É a escolha ideal quando se sabe previamente o conjunto de itens a ser percorrido.
- O laço **`while`**, que executa um bloco de código repetidamente enquanto uma condição específica for avaliada como verdadeira. É perfeito para situações em que o número de repetições não é conhecido de antemão.

Laço FOR

O for permite percorrer os itens de uma coleção (Lista, String, Tuplas, Dicionários) e, para cada um deles, executar um bloco de código. O for é utilizado para percorrer ou iterar sobre uma sequência de dados, executando um conjunto de instruções em cada item. A sintaxe básica do for pode ser escrita da seguinte forma:

```
for <variavel_de_iteracao> in <sequencia>:
    # Bloco de código que será executado para cada item
    # A <variavel_de_iteracao> guarda o item atual da sequência
```

Veja o exemplo onde temos uma lista, uma estrutura de dados que já veremos com mais detalhes em breve. Será utilizado o laço de repetição FOR para percorrer cada elemento dessa lista e imprimir seu valor na tela de saída.

Copie e Teste!

```
lista = [10,20,30,40,50]
for item in lista:
    print(item)
```

Saída Esperada

```
10
20
30
40
50
```

Caso Prático

Imagine que você tem uma lista de números inteiros, e deseja dobrar o valor de cada elemento e adicioná-lo a uma nova lista. Nesse caso, podemos criar uma nova lista vazia, e para cada elemento da nossa lista, dobramos o seu valor e o adicionamos à nova lista.

Copie e Teste!

```
lista = [10,20,30,40,50]
listaDobro = []
for item in lista:
  listaDobro.append(item * 2)

print("========Saída===========================")
print("LISTA: ", lista)
print("LISTA COM OS VALORES DOBRADOS: ",listaDobro)
print("========================================")
```

Saída Esperada

```
========Saída===========================
LISTA: [10, 20, 30, 40, 50]
LISTA COM OS VALORES DOBRADOS: [20, 40, 60, 80, 100]
========================================
```

A saída confirma que o laço for percorreu cada item da lista original. Para cada elemento a operação matemática (item * 2) foi executada e o resultado foi adicionado à listaDobro com o método `.append()`.

Caso Prático

Imagine que você tem uma lista de supermercado e deseja imprimir todos os elementos dessa lista no seguinte formato “PRODUTO: NOME”.

Copie e Teste!

```
listaSupermercado = ["Arroz","Farinha","Açúcar"]

for produto in listaSupermercado:
    print("PRODUTO :", produto)
```

Saída Esperada

```
PRODUTO: Arroz
PRODUTO: Farinha
PRODUTO: Açucar
```

Conhecendo um pouco mais!

Material Extra
Conheça um pouco mais a estrutura de repetição FOR em Python. .

https://www.hashtagtreinamentos.com/estruturas-de-repeticao-python?gad_source=1&gclid=CjwKCAiAxea5BhBeEiwAh4t5KzH-d_Snz95z0Cs2p9iHtLlRVVfvFgpjE74lRQGpGaN2d66jOFqS1RoCljIQAvD_BwE

ou aponte a câmera do seu smartphone para o Qr Code ao lado

A função `RANGE()` é uma função nativa do Python que retorna uma sequência de números, começando em 0 por padrão e incrementa em 1 (por padrão) e termina em um número especificado. Em outras palavras, a função `RANGE()` cria uma lista e retorna essa lista que podemos usar em uma estrutura de repetição para executar uma tarefa inúmeras vezes.

Caso Prático

Imagine que você queira executar uma tarefa 5 vezes, utilizando for, mas você não tem uma lista para percorrer. Você pode fazer isso usando o `RANGE()`.

Copie e Teste!

```
for i in range(5):
    print("Executando alguma coisa...")
```

Saída Esperada

```
Executando alguma coisa...
Executando alguma coisa...
Executando alguma coisa...
Executando alguma coisa...
Executando alguma coisa...
```

Conhecendo um pouco mais!

Material Extra
Conheça um pouco mais a estrutura de repetição FOR com a função `RANGE()`.

https://www.youtube.com/watch?v=55rOjj6kEck

ou aponte a câmera do seu smartphone para o Qr Code ao lado

Laço WHILE

Utilizando o while, podemos executar de forma repetida um conjunto de instruções enquanto uma condição for verdadeira. No exemplo à seguir, a mensagem executando... é impressa na tela até que a condição contador < 6 seja satisfeita.

Copie e Teste!

```
contador = 1

while contador <= 5:
    print("Executando ",contador, "Vez")
    contador += 1
```

Saída Esperada

```
Executando  1 Vez
Executando  2 Vez
Executando  3 Vez
Executando  4 Vez
Executando  5 Vez
```

Observe que toda vez que o laço de repetição é executado, o contador é incrementado, ou seja, uma unidade é somada ao contador. Isso é importante, pois temos que garantir que uma hora a condição será falsa para finalizar a repetição. Sendo assim, o laço de repetição será executado 5 vezes.

1.3.4 Aplicando seus Conhecimentos

1. Escreva um programa que leia a idade de uma pessoa, e informe se a pessoa é maior ou menor de idade.

2. Escreva um programa que, dados 2 números diferentes (a e b), encontre o menor deles.

3. Leia dois números do teclado e efetue a soma desses números. Caso o valor somado seja maior que 20, este deverá ser impresso na tela, somando-se a ele mais 8, caso o valor somado seja menor ou igual a 20, este deverá ser apresentado subtraindo-se 5.

4. Faça um programa que pergunte a idade e peso para uma pessoa e diga se ela pode doar sangue ou não. Para doar sangue é necessário

   - Pesar mais de 50 kg
   - Ter entre 16 e 69 anos

5. Faça um programa que pede duas notas de um aluno. Em seguida ele deve calcular a média do aluno e dar o seguinte resultado.

   - A mensagem ”Aprovado”, se a média alcançada for maior ou igual a sete
   - A mensagem ”Reprovado”, se a média for menor do que sete

6. Utilize laços de repetição para elaborar um código cuja entrada sejam 5 números inteiros, e a saída retorne a quantidade de números pares e a quantidade de números ímpares.

    - Use variáveis para acumular a quantidade de números pares e a quantidade de números ímpares.
    - Use o operador Módulo (`%`) para saber se um número é par.
      Exemplo: 7 % 2 = 1 (Número ímpar), 8 % 2 = 0 (Número par).

7. Utilize laços de repetição combinandos com a função `RANGE()`

1.4 Estruturas de Dados Essenciais

Até este ponto, focamos em como armazenar valores individuais em variáveis, como um único número ou uma string. No entanto, a maioria dos problemas do mundo real exige o gerenciamento de **conjuntos de dados**: uma lista de alunos, as características de um produto ou as coordenadas de um mapa. Armazenar cada um desses valores em uma variável separada seria ineficiente e impraticável.

É para resolver essa questão que a programação nos oferece as **estruturas de dados**. Conforme demonstrado em obras práticas como a de Matthes (2023), essas estruturas são a base para a construção de projetos complexos, permitindo-nos modelar coleções de informações de forma organizada e acessível. Uma estrutura de dados, portanto, é um formato especializado para organizar, processar e armazenar múltiplos elementos de forma coesa.

A escolha da estrutura correta é uma decisão fundamental no desenvolvimento de um software. Como aponta Ramalho (2015), entender as características e o uso idiomático de cada estrutura de dados em Python é o que diferencia um programador iniciante de um programador fluente, impactando diretamente a performance e a clareza do código.

Python oferece diversas estruturas de dados nativas, cada uma com características e casos de uso específicos. Nesta seção, exploraremos as quatro mais essenciais:

- **Listas:** Coleções ordenadas e mutáveis de elementos, ideais para quando você precisa de uma sequência flexível que pode ser alterada.
- **Tuplas:** Similares às listas, são coleções ordenadas, porém **imutáveis**, usadas para garantir a integridade dos dados.
- **Dicionários:** Estruturas que organizam dados em pares de `chave-valor`, permitindo uma associação lógica entre informações.
- **Conjuntos (Sets):** Coleções não ordenadas que armazenam apenas **elementos únicos**, sendo extremamente eficientes para operações matemáticas de conjuntos.

1.4.1 Listas: Coleções Ordenadas e Mutáveis

Listas são estruturas de dados capazes de armazenar múltiplos elementos. Uma lista é uma coleção de elementos, podendo ser do tipo String, números ou até objetos (Matthes, 2023). Para se declarar uma lista em Python basta colocar os elementos separados por vírgulas dentro de colchetes [ ]. Veja um exemplo abaixo, na qual temos uma lista, representando uma lista de supermercado com vários elementos separados por vírgulas.

Copie e Teste!

```
#Criando uma lista
lista = ["Arroz","Feijão", "Açúcar", "Farinha", "Óleo"]

#Imprimindo a lista
print(lista)
```

Saída Esperada

```
["Arroz","Feijão", "Açúcar", "Farinha", "Óleo"]
```

Acessando elementos de uma lista

Para acessar um elemento de uma lista podemos fazer isso pelo seu índice, ou posição na lista. Lembrando que essa contagem começa a partir do zero. Observe o exemplo abaixo, onde é impresso na tela o item "Arroz"e "Feijão" que estão nas posições 0 e 1 respectivamente.

Copie e Teste!

```
#Criando uma lista
lista = ["Arroz","Feijão", "Açúcar", "Farinha", "Óleo"]

#Acessando um elemento da lista pelo índice
print("Item :", lista[0])
print("Item :", lista[1])
```

Saída Esperada

```
Item: Arroz
Item: Feijão
```

Alterando elementos de uma lista

Podemos alterar o valor de um elemento da lista, para isso usamos o operador "=" para um determinado elemento, observe que obtemos um elemento pela posição. Observe no exemplo abaixo que mudamos o elemento da primeira posição que era "Arroz" para "Leite".

Copie e Teste!

```
#Criando uma lista
lista = ["Arroz", "Feijão", "Açúcar"]

#Acessando o elemento pelo índice e atualizando seu valor
lista[0] = "Leite"
```

Saída Esperada

```
lista = ["Leite", "Feijão", "Açucar"]
```

Removendo elementos de uma lista

Uma lista é uma estrutura de dados mutável, ou seja, pode ter seus valores alterados. Podemos remover um item utilizando o valor que o item representa dentro da lista através do operador `remove()`, ou ainda, baseado na posição que ele ocupa através do operador `del`.

Copie e Teste!

```
#Criando uma lista
lista = ["Arroz", "Feijão", "Açúcar", "Farinha", "Óleo"]
print("LISTA INICIAL: ", lista)

#Removendo um elemento da lista pelo índice
del lista[0]
print("LISTA FINAL: ", lista)
```

Saída Esperada

```
LISTA INICIAL: ["Arroz", "Feijão", "Açúcar", "Farinha", "
   Óleo"]
LISTA FINAL: ["Feijão", "Açúcar", "Farinha", "Óleo"]
```

Foi utilizado o operador `del` para remover o primeiro elemento da lista, identificado pela posição 0. Como pode ser observado, a lista o primeiro elemento desaparece e a lista final passa a ter apenas 4 elementos.

Adicionando elementos de uma lista

Podemos também adicionar elementos a uma lista através do método `append()`, o qual adiciona o elemento no final da lista. Para ilustrar seu comportamento temos o exemplo abaixo de uma lista de amigos inicialemnte vazia.

Copie e Teste!

```
#Criando a lista vazia
amigos = []
#Adicionando elementos na lista
amigos.append("JOÃO")
amigos.append("MARIA")
amigos.append("ANA")

#Imprimindo lista de amigos
print("LISTA DE AMIGOS :", amigos)
```

Saída Esperada

```
LISTA DE AMIGOS: ["JOÃO", "MARIA", "ANA"]
```

Ordenando Listas

Também é possível ordenar os elementos dentro listas, de forma crescente ou decrescente. No caso de listas cusjo elementos são números, é possível ordenador do maior para o menor e vice-versa, enquanto que listas que envolvem string é possível organizar em ordem alfabética direta e invertida. Para ordenar uma lista é utilizado o método `sort()`.

Copie e Teste!

```
#Criando uma lista
amigos = []
amigos.append("JOÃO")
amigos.append("MARIA")
amigos.append("ANA")
#Imprimindo a lista
print("LISTA DE AMIGOS :", amigos)

#Ordenando a lista
amigos.sort()
#Imprimindo a lista ordenada de forma crescente
print("LISTA DE AMIGOS ORDENADA :", amigos)
```

Saída Esperada

```
LISTA DE AMIGOS: ["JOÃO", "MARIA", "ANA"]
LISTA DE AMIGOS ORDENADA: ["ANA", "JOÃO", "MARIA"]
```

1.4.2 Tuplas: Coleções Ordenadas e Imutáveis

Uma tupla em PYTHON é uma estrutura de dados similar à uma lista, na qual temos uma coleção de elementos indexados. Porém, diferentemente das listas, uma tupla é uma estrutura imutável, ou seja, após ser criada, não podemos alterar seus elementos, nem adicionar ou remover elementos da mesma.

A sintaxe para se criar uma tupla em PYTHON é similar as listas, com os elementos separados por virgula mas limitados por parenteses ( ), conforme o exemplo à seguir:

Copie e Teste!

```
minhaTupla = ("André", "João", "Maria")
print(minhaTupla)
```

Saída Esperada

```
("André", "João", "Maria")
```

Acessando elementos de uma tupla

Assim como na estrutura de lista, ao se usar tuplas, podemos acessar seus elementos através da posição. No exemplo abaixo será realizado o acesso ao primeiro elemento da tupla, lembrando que a contagem no Python sempre começa do índice0.

Copie e Teste!

```
minhaTupla = ("André", "João", "Maria")
print("Primeiro elemento da tupla :", minhaTupla[0])
```

Saída Esperada

```
Primero elemento da tupla: André
```

Tentando alterar elementos de uma tupla

A principal característica que define uma tupla é a sua imutabilidade. Uma vez criada, ela se torna uma coleção de dados ”somente leitura”: não podemos alterar, adicionar ou remover seus elementos.

Essa propriedade é uma garantia de segurança, assegurando que dados importantes não sejam modificados acidentalmente em outra parte do programa. Mas o que acontece quando tentamos quebrar essa regra? O interpretador Python nos impedirá com um erro claro.

O exemplo a seguir tenta deliberadamente modificar o segundo elemento da tupla de ”João”para ”José”.

Copie e Teste!

```
minhaTupla = ("André", "João", "Maria")
minhaTupla[1] = "José"
```

Saída Esperada

```
Traceback (most recent call last):
  File "C:\Users\PC\PycharmProjects\CITHA\TUPLAS\Tuplas.py",
    line 4, in <module>
    minhaTupla[1] = "Jose"
    ~~~~~~~~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

A ”Saída Esperada”nos mostra um Traceback, que é o relatório de erro do Python:

1. `File "..."`: Informa o arquivo e a linha exata onde o erro ocorreu (linha 4).
2. `minhaTupla[1] = "Jose"`: Mostra o código problemático.
3. `TypeError: 'tuple' object does not support item assignment`:
   Esta é a mensagem mais importante. Ela nos diz que houve um ”Erro de Tipo”(TypeError) porque objetos do tipo ’tupla’ (’tuple’ object) não suportam atribuição de item (does not support item assignment), ou seja, não permitem que seus elementos sejam alterados.

Este erro não é uma falha, mas sim o Python aplicando corretamente e explicando a regra de imutabilidade das tuplas.

Fique Alerta!

Pense nas tuplas como uma forma de ”proteger”seus dados. Por serem imutáveis, uma vez que você cria uma tupla, seus elementos não podem ser alterados, adicionados ou removidos. Isso oferece uma garantia de segurança, tornando-as a escolha perfeita para armazenar informações que devem permanecer constantes e íntegras durante toda a execução do programa, como coordenadas geográficas ou dados de configuração.

1.4.3 Dicionários: Estruturas Chave-Valor

O Dicionário é uma uma estrutura de dados mais completa, e representa coleções de dados que contém na sua estrutura um conjunto de pares chave/valor, nos quais cada chave individual tem um valor associado.

Para se criar um dicionário utiliza-se uma sintaxe um pouco mais elaborada do que as listas e tuplas. Os elementos do dicionário continuam sendo separados por vírgula, mas agora são delimitados por chaves . Além disso, cada elemento do dicionário é composto por 2 subelementos separados por dois pontos fazendo a separação entre chave e valor.

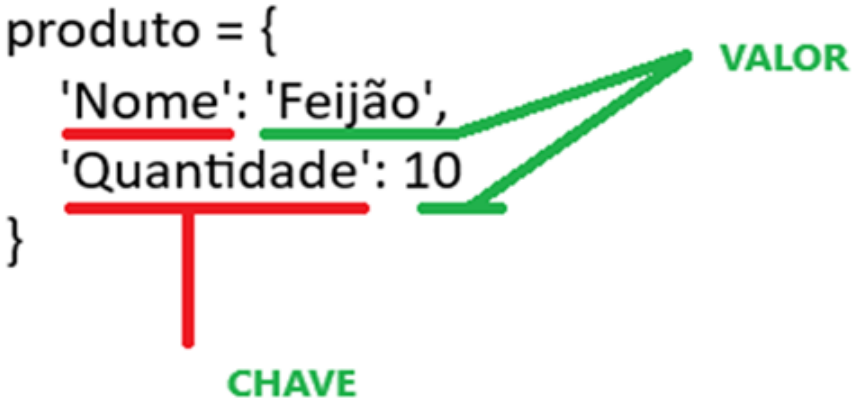

Figura 1.20: Sintaxe de um dicionário
Fonte: Elaborada pelos autores

O exemplo represetando pela figura 1.20 ilustra a estrutura de um dicionário onde temos uma variável chamada produto, que possuí algumas características que são as chaves. A chave `Nome` possuí o valor **Feijão**, enquanto que a chave `Quantidade` possuí o valor **10**.

Acessando elementos de um dicionário

Para acessar os elementos de um dicionário, podemos usar a sua chave. Veja o exemplo abaixo, onde iremos pegar o valor do Nome do produto.

Copie e Teste!

```
produto = {
    'Nome': 'Feijão',
    'Quantidade': 10
}

nome = produto['Nome']
quantidade = produto['Quantidade']
print("Nome : ", nome)
print('Quantidade : ', quantidade)
```

Saída Esperada

```
Nome: Feijão
Quantidade: 10
```

Alterando elementos de um dicionário

Para alterar o valor de um elemento em um dicionário, podemos realizar uma operação similar à apresentada nas listas, porém no lugar de acessar o elemento pelo seu índice, podemos usar a chave. Veja o exemplo abaixo, onde vamos alterar a quantidade do produto.

Copie e Teste!

```
produto = {
   'Nome': 'Feijão',
   'Quantidade': 10
}

#Alterando o valor
produto['Quantidade'] = 100

nome = produto['Nome']
quantidade = produto['Quantidade']
print("Nome : ", nome)
print('Quantidade : ', quantidade)
```

Saída Esperada

```
Nome: Feijão
Quantidade: 100
```

Verificando elementos de um dicionário

Utilizando dicionários é possível organizar todos os elementos em uma lista utilizando a estrutura de repetição (for). Observe que agora o nosso dicionário tem uma lista de chaves e valores que é retornada pela função `items()`, a qual retorna a chave e o valor de cada elemento do nosso dicionário. No exemplo abaixo temos duas chaves, onde cada uma dessas chaves são definidas através de uma lista:

Copie e Teste!

```
produto = {
  'Nome': ['Feijão','Arroz', 'Farinha'],
  'Quantidade': [10,10, 100]
}

# Verificando os itens do dicionário
for chave, valor in produto.items():
  print(chave, valor)

# Verificando os itens da chave nome
for item in produto['Nome']:
  print(item)
```

Saída Esperada

```
Nome: ["Feijão", "Arroz", "Farinha"]
Quantidade: [10, 10, 100]

Feijão
Arroz
Farinha
```

Verificamos que é possível acessar os itens relativos à chave escolhida, no caso os elementos da lista `'Nome'`.

Agora imagine que você precisa imprimir todos os produtos e suas respectivas quantidades. Para isso vamos usar a função `zip()`, que tem como finalidade juntar duas listas com suas respectivas posições e retorna os valores correspondentes, no nosso exemplo os valores correspondentes das listas `'Nome'` e `'Quantidade'`.

Copie e Teste!

```
produto = {
    'Nome': ['Feijão','Arroz', 'Farinha'],
    'Quantidade': [10,10, 100]
    }

for nome,quantidade in zip(produto['Nome'],produto['Quantidade']):
     print("Produto:",  nome, "Quantidade : ",quantidade )
```

Saída Esperada

```
Produto: Feijão Quantidade :  10
Produto: Arroz Quantidade :  10
Produto: Farinha Quantidade :  100
```

A saída demonstra o poder da função `zip()`. Ela atuou como um zíper, unindo os elementos das listas `'Nome'` e `'Quantidade'` em pares, um a um. O laço for então percorreu esses pares, desempacotando-os nas variáveis nome e quantidade a cada iteração, permitindo-nos exibir os dados de forma organizada. Essa é uma técnica extremamente útil para processar múltiplas sequências de dados que se correspondem.

1.4.4 Conjuntos (Sets): Coleções de Itens Únicos

Um conjunto é uma coleção de elementos **únicos** e **não ordenada**. Essa característica os torna a ferramenta perfeita para duas tarefas principais: remover elementos duplicados de outras coleções e realizar operações matemáticas de conjuntos, como união, interseção e diferença. As principais características de um conjunto são:

- **Itens Únicos:** Um conjunto não permite elementos duplicados. Se você tentar adicionar um item que já existe, nada acontecerá.
- **Não Ordenados:** Os itens em um conjunto não têm uma posição fixa ou um índice. Por isso, você não pode acessar seus elementos com `conjunto[0]`.
- **Mutáveis:** Você pode adicionar ou remover itens de um conjunto após sua criação.

Você pode criar um conjunto a partir de uma lista para remover duplicatas ou declará-lo diretamente com chaves `{}`. Para criar um conjunto vazio, é obrigatório usar a função `set()`, pois `{}` cria um dicionário vazio.

Copie e Teste!

```python
numeros = [1, 2, 2, 3, 4, 4, 4]
numeros_unicos = set(numeros)

print(f"Lista original: {numeros}")
print(f"Conjunto de numeros unicos: {numeros_unicos}")
```

Saída Esperada

```
Lista original: [1, 2, 2, 3, 4, 4, 4]
Conjunto de numeros unicos: {1, 2, 3, 4}
```

Observe atentamente a `'Saída Esperada'`. A Lista original continha vários números repetidos (o 2 e o 4). Ao aplicarmos a função set(), o resultado é um Conjunto que, por definição, armazena apenas os elementos únicos. Todas as duplicatas foram automaticamente removidas. Note também que a representação de um conjunto utiliza chaves , reforçando que se trata de uma estrutura de dados diferente das listas [].

Manipulando elementos em um conjunto

Para manipular os elementos dentro de um conjunto, utilizamos os métodos `add()` para adicionar novos elementos, e o método `remove()` para remover elementos existentes. Como os elementos de um conjunto são únicos, adicionar um elementos repetido não surte qualquer efeito, conforme pode ser observado no exemplo à seguir:

Copie e Teste!

```
participantes = {"Ana", "Bia", "Carlos"}
print(f"Participantes iniciais\n {participantes} \n")

# Adicionando um novo participante
participantes.add("Daniel")
print(f"Apos adicionar 'Daniel'\n {participantes} \n")

# Tentar adicionar um participante que ja existe
participantes.add("Ana")
print(f"Apos tentar readicionar 'Ana'\n {participantes} \n")

# Removendo um participante
participantes.remove("Bia")
print(f"Apos remover 'Bia'\n {participantes}")
```

Saída Esperada

```
Participantes iniciais
{'Bia', 'Carlos', 'Ana'}

Apos adicionar 'Daniel'
{'Bia', 'Daniel', 'Carlos', 'Ana'}

Apos tentar readicionar 'Ana'
{'Bia', 'Daniel', 'Carlos', 'Ana'}

Apos remover 'Bia'
{'Daniel', 'Carlos', 'Ana'}
```

Operações com elementos de um conjunto

A verdadeira força dos conjuntos aparece quando realizamos operações lógicas entre eles, o que é extremamente rápido e eficiente.

Caso Prático

Uma escola de tecnologia que atualmente oferece cursos de ”Python Avançado”e ”Lógica Básica”, quer analisar qual o total de alunos, quanto fazem mais de um curso simultaneamente e quais escolheram ”Python Avançado”antes de cursar ”Lógica Básica”.

Copie e Teste!

```
turma_python = {"Ana", "Beto", "Carla", "Daniel"}
turma_logica = {"Carla", "Daniel", "Eva", "Fabio"}

# 1. União (|): Quem sao todos os alunos, sem repeticao?
todos_alunos = turma_python | turma_logica
print(f"Todos os alunos\n {todos_alunos} \n")

# 2. Interseção (&): Quais alunos estao em AMBAS as turmas?
alunos_em_ambas = turma_python & turma_logica
print(f"Alunos em ambas as turmas\n {alunos_em_ambas} \n")

# 3. Diferença (-): Quais alunos fazem Python, mas não fazem
   Lógica?
alunos_so_python = turma_python - turma_logica
print(f"Alunos que so fazem Python\n {alunos_so_python}")
```

Saída Esperada

```
Todos os alunos
{'Fabio', 'Eva', 'Carla', 'Beto', 'Daniel', 'Ana'}

Alunos em ambas as turmas
{'Carla', 'Daniel'}

Alunos que so fazem Python
{'Beto', 'Ana'}
```

Fique Alerta!

Além de remover duplicatas, os conjuntos são a estrutura de dados mais eficiente em Python para verificar se um item pertence a uma coleção. A operação `item in meu_conjunto` é muito mais rápida do que `item in minha_lista`, especialmente para grandes volumes de dados.

Conhecendo um pouco mais!

Material Extra
Conheça um pouco mais sobre dicionários e conjuntos em python.

`https://docs.python.org/pt-br/3/tutorial/datastructures.html#`

ou aponte a câmera do seu smartphone para o Qr Code ao lado

1.4.5 Aplicando seus conhecimentos

1. **Gerenciador de Tarefas (To-Do List):** Crie um programa que funcione como um gerenciador de tarefas. O programa deve permitir que o usuário:

   (a) Adicione uma nova tarefa à lista.

   (b) Visualize todas as tarefas da lista, em ordem alfabética.

   (c) Marque uma tarefa como concluída (removendo-a da lista).

   (d) O programa deve continuar em execução até que o usuário escolha a opção ”sair”.

   Dicas: Use um laço `while True` para manter o programa rodando. Peça ao usuário para digitar comandos como ”adicionar”, ”remover”, ”ver”ou ”sair”. Use condicionais (`if/elif/else`) para executar a ação correta. Lembre-se dos métodos `.append()`, `.remove()` e `.sort()`.

2. **Filtrando Dados Numéricos:** Seja uma lista de valores referentes a um certo processo `dados = [10, -5, 22, 0, 15, 8, -1, -12]`, escreva um algoritmo que crie duas novas listas, uma contendo apenas os números positivos e outra contendo apenas os números negativos. Ao final, imprima as duas novas listas.

   Dica: Percorra a lista original com um laço `for` e use um `if` para decidir em qual nova lista cada número deve ser adicionado.

3. **Armazenando Coordenadas Geográficas:** Um sistema de mapeamento precisa armazenar um conjunto de coordenadas geográficas (latitude, longitude) que não devem ser alteradas. Crie uma lista de tuplas, onde cada tupla contém uma latitude e uma longitude, por exemplo:

   `locais = [(-23.5505, -46.6333), (-22.9068, -43.1729)]`.

   Depois, escreva um código que percorra essa lista e imprima apenas as coordenadas de locais no Hemisfério Sul (ou seja, com latitude negativa).

4. **Dados de Configuração:** As tuplas são excelentes para guardar dados de configuração que não devem mudar. Crie uma tupla para armazenar as configurações de um servidor, como `config_servidor = ("192.168.1.1", 8080, "admin")`. Desempacote essa tupla em três variáveis separadas: `ip`, `porta` e `usuario`, e imprima cada uma delas.

5. **Contagem de Palavras:** Escreva um programa que conte a frequência de cada palavra em uma frase. Para a frase `"O rato roeu a roupa do rei de roma"`, seu programa deve gerar um dicionário como saída, informando quantas vezes cada palavra apareceu.

   Dica: Use o método `.split()` para transformar a frase em uma lista de palavras. Percorra essa lista com um laço `for`. Para cada palavra, verifique se ela já está no dicionário. Se estiver, incremente seu valor. Se não, adicione-a com o valor 1.

6. **Cadastro de Alunos e Notas:** Crie um programa que simule um pequeno cadastro de alunos. O programa deve usar um dicionário onde as chaves são os nomes dos alunos e os valores são listas com suas notas.

   (a) Permita que o usuário adicione um novo aluno e suas notas.

(b) Crie uma função que receba o dicionário e o nome de um aluno, e retorne a média de suas notas.

(c) Ao final, imprima a média de um aluno específico.

7. **Gerenciador de Inscrições em Eventos:** Imagine que você está organizando dois eventos: um Workshop de Python e uma Palestra de IA. Crie dois conjuntos para armazenar os nomes dos participantes inscritos em cada evento. Seu programa deve ser capaz de responder:

   (a) Quais participantes estão inscritos em **ambos** os eventos? (Interseção)

   (b) Quais são todos os participantes únicos, somando os dois eventos? (União)

   (c) Quais participantes se inscreveram **apenas** no Workshop de Python? (Diferença)

8. **Verificação de Ingredientes Únicos:** Um chef de cozinha tem uma lista de ingredientes para uma receita, mas alguns podem estar repetidos: `ingredientes=["farinha","açúcar","ovo","leite","açúcar","fermento","ovo"]`. Use um conjunto para descobrir quantos ingredientes **únicos** a receita possui e imprima a lista de ingredientes únicos.

1.5 Considerações do Módulo 1

Conhecendo um pouco mais!

Acessando os Materiais Práticos do Curso
Agora que você concluiu os fundamentos, é hora de colocar a mão na massa. Para facilitar seus estudos e a execução dos exercícios, todos os exemplos e conjuntos de dados (*datasets*) utilizados no livro estão disponíveis em nosso repositório oficial do projeto no GitHub.

`https://github.com/CITHA-AM/Python`

Sugerimos duas maneiras de utilizar esses materiais:

Usando os Notebooks no Google Colab: A forma mais simples e interativa de acompanhar o curso é através dos `notebooks` (`.ipynb`). A grande vantagem desta abordagem é que os notebooks já estão configurados para carregar os *datasets* necessários diretamente da internet, sem a necessidade de downloads. Basta abri-los no Google Colab e começar a executar o código.

Trabalhando Localmente com os Scripts: Se você prefere trabalhar em um ambiente local (como o PyCharm), a melhor abordagem é baixar o conteúdo do repositório. A maneira mais fácil é ir à página principal, clicar no botão verde <> **Code** e selecionar **Download ZIP**.

Fique Alerta!

Atenção: Para que os scripts (`.py`) funcionem corretamente, o arquivo de script e seu respectivo *dataset* (ex: `dados.csv`) devem estar na mesma pasta. Por isso, ao baixar o ZIP, mantenha a estrutura de pastas original.

Parabéns por concluir o primeiro módulo! Ao longo desta jornada inicial, você construiu uma base sólida que é o alicerce de toda a programação em Python. Partimos da sintaxe e dos conceitos mais básicos da linguagem, aprendemos a manipular dados individuais com variáveis, tipos e operadores e, em seguida, descobrimos como controlar o fluxo de um programa com estruturas condicionais e laços de repetição. Por fim, exploramos as estruturas de dados essenciais — listas, tuplas, dicionários e conjuntos — que nos permitem organizar e gerenciar coleções de informações de maneira eficiente.

É crucial entender que esses conceitos não são apenas comandos isolados; eles são o alfabeto e a gramática da lógica de programação. Com as ferramentas que você adquiriu neste módulo, você já é capaz de escrever programas completos que resolvem problemas reais, automatizam tarefas e processam dados de forma estruturada. O passo mais importante agora é a **prática contínua**. Desafie-se com novos problemas, modifique os exemplos do livro e não tenha medo de experimentar. A programação é uma habilidade que se fortalece com a aplicação.

Agora que você já sabe como construir a lógica interna de um programa, o próximo módulo o levará a um novo patamar de organização e poder. Vamos explorar como encapsular blocos de código em **funções**, tornando nossos programas mais limpos, reutilizáveis e fáceis de manter. Além disso, aprenderemos a interagir com o mundo exterior, **lendo e escrevendo em arquivos**, o que permitirá que nossos programas salvem seus resultados e trabalhem com dados de fontes externas. Continue com essa dedicação, pois os conceitos a seguir expandirão enormemente o que você é capaz de criar.

Capítulo 2

Modularização e Persistência de Dados

Iniciando o diálogo...

No módulo anterior, você aprendeu os blocos de construção da programação. Agora, vamos usá-los para criar estruturas mais inteligentes e duradouras. Este capítulo foca em duas ideias poderosas: organizar seu código em blocos reutilizáveis com **Funções** e ensinar seu programa a lembrar informações salvando-as em **Arquivos**. Dominar isso é o que transforma scripts simples em aplicações robustas e funcionais.

Suponha que voce pretende construir uma casa, não existe um manual específico com milhões de passos sequenciais. O projeto é dividido em tarefas especializadas, como ”fazer a fundação”ou ”instalar a parte elétrica”. Na programação, fazemos o mesmo usando **Funções**. Cada função é como uma equipe especializada que realiza uma tarefa específica, o que torna nosso código mais limpo, organizado e reutilizável. Em vez de reescrever o mesmo código várias vezes, você simplesmente ”chama”a função sempre que precisar.

Da mesma forma, uma casa não tem utilidade se tudo dentro dela some quando você sai. Nossos programas precisam de **persistência de dados** para serem úteis, ou seja, a capacidade de salvar informações permanentemente, mesmo após o término da execução. Vamos explorar como ler e escrever desde simples arquivos de texto (`.txt`) até formatos de dados estruturados como `CSV` e `JSON`, que são essenciais para a troca de informações na tecnologia hoje.

Neste capítulo, você dará o próximo passo para se tornar um programador fluente, aprendendo a construir programas que não são apenas lógicos, mas também bem-estruturados e com memória.

2.1 Funções: Organizando seu Código

No início deste capítulo, comparamos a criação de um programa à construção de uma casa. Vimos que seria impraticável seguir um único manual gigante com milhões de passos sequenciais. Em vez disso, o trabalho é dividido em partes lógicas e gerenciáveis: a equipe da fundação, os eletricistas, os encanadores. Cada equipe é especialista em uma tarefa, e seu trabalho pode ser solicitado sempre que necessário. Esta é a essência da **modularização**, e a principal ferramenta para aplicá-la em Python são as **funções**.

Pense em uma função como uma ”receita”ou um ”plano de ação”nomeado que realiza uma tarefa específica e bem definida. Em vez de reescrever todos os passos para ”calcular a média de notas de um aluno”toda vez que for preciso, você pode criar uma única função

chamada `calcular_media()`. A partir daí, sempre que necessitar dessa operação, você simplesmente ”chama”a função pelo nome, e ela executa a tarefa para você.

2.1.1 A Necessidade de Abstração

Até agora, nossos programas seguiram uma lógica linear. Se precisássemos realizar a mesma operação em três partes diferentes do código, a solução mais óbvia seria copiar e colar o mesmo bloco de instruções.

Imagine que você está desenvolvendo um sistema para monitorar a saúde de uma área de reflorestamento na Amazônia. Você precisa coletar dados de um igarapé próximo a partir de três sensores diferentes:

- umidade do solo
- temperatura do ar
- qualidade da água

Para cada sensor, você precisa ler o dado, verificar se ele está dentro de uma faixa aceitável e, caso não esteja, registrar um alerta. Sem o conceito de funções, seu código poderia se parecer com isto:

Copie e Teste!

```
# --- Bloco para o Sensor de Umidade ---
umidade = 45.7
print(f"Lendo dado de umidade: {umidade}%")
if not (30 <= umidade <= 70):
    print("ALERTA: Umidade do solo fora do padrão!")
else:
    print("Umidade do solo OK.")
print("-" * 20)

# --- Bloco para o Sensor de Temperatura ---
temperatura = 31.5
print(f"Lendo dado de temperatura: {temperatura}°C")
if not (22 <= temperatura <= 30):
    print("ALERTA: Temperatura do ar fora do padrão!")
else:
    print("Temperatura do ar OK.")
print("-" * 20)

# --- Bloco para o Sensor de pH da Água ---
ph_agua = 5.8
print(f"Lendo dado de pH da água: {ph_agua}")
if not (6.5 <= ph_agua <= 8.5):
    print("ALERTA: pH da água fora do padrão!")
else:
    print("pH da água OK.")
print("-" * 20)
```

Saída Esperada

```
Lendo dado de umidade: 45.7\%
Umidade do solo OK.
--------------------
Lendo dado de temperatura: 31.5°C
ALERTA: Temperatura do ar fora do padrão!
--------------------
Lendo dado de pH da água: 5.8
ALERTA: pH da água fora do padrão!
--------------------
```

Agora, imagine que você descobre um erro na lógica ou decide mudar a forma como os alertas são exibidos. Você teria que encontrar e corrigir todas as cópias espalhadas pelo seu programa. Esse método, além de cansativo, é uma fonte perigosa de erros e vai contra um princípio fundamental da programação:

DRY - Don't Repeat Yourself

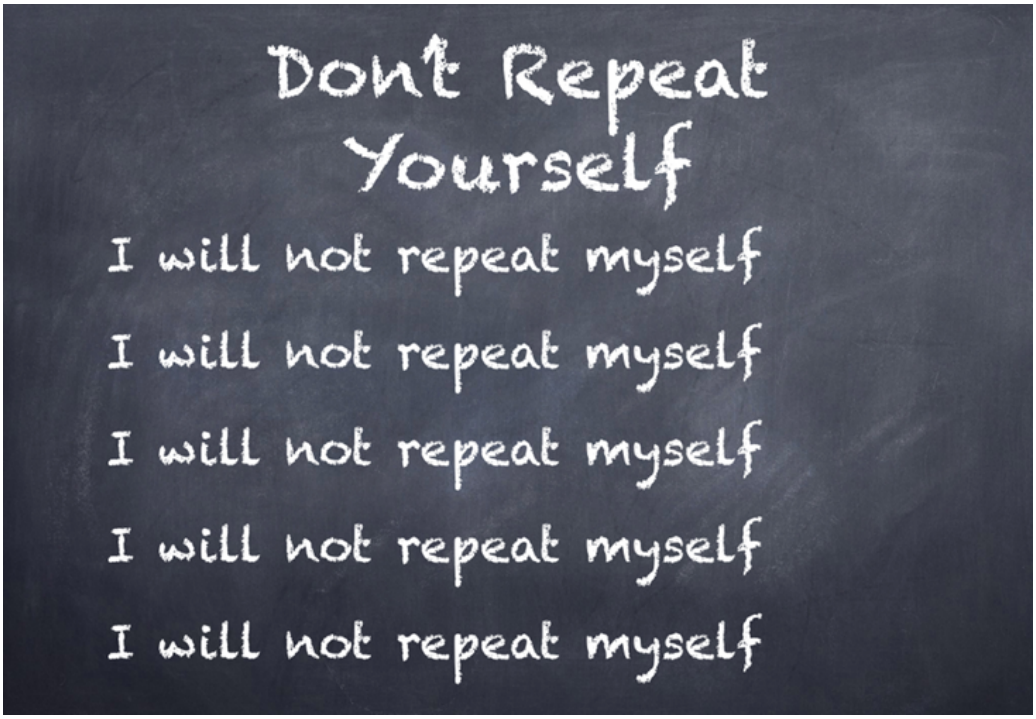

Figura 2.1: Exemplo do que evitar utilizando o conceito DRY
Fonte: Peakd, 2019

Repetir código é considerado uma má prática, pois torna os programas difíceis de ler, depurar e manter. As funções são a nossa principal ferramenta para seguir o princípio DRY e praticar a abstração.

Abstração é a ideia de esconder os detalhes complexos de uma implementação por trás de uma interface simples. Quando você usa a função `print()`, por exemplo, não precisa saber como ela interage com o sistema operacional para fazer os caracteres aparecerem na tela; você

apenas confia que ela fará seu trabalho. Da mesma forma, ao criar suas próprias funções, você encapsula a lógica — o "como fazer"— dentro de um bloco nomeado, focando apenas no "o que a função faz". Como discutido por Menezes (2019), o domínio dessas estruturas é um passo essencial no aprendizado de algoritmos e lógica.

Fique Alerta!

Abstração não é sobre remover complexidade, é sobre **organizá-la**. A complexidade de verificar um sensor ainda existe, mas ela está contida dentro de uma função, como uma ferramenta em uma caixa. Você não precisa ver todas as engrenagens da ferramenta para usá-la, apenas saber qual botão apertar.

Caso Prático

Vamos criar uma função chamada `verificar_sensor`. Ela vai receber o nome do sensor, seu valor, e os limites mínimo e máximo aceitáveis. A função fará todo o trabalho de verificação e impressão.

Copie e Teste!

```
def verificar_sensor(nome_sensor, valor, unidade, min_aceitavel,
    max_aceitavel):
    """
    Verifica o dado de um sensor e imprime seu status.
    """
    print(f"Lendo dado de {nome_sensor}: {valor}{unidade}")
    if not (min_aceitavel <= valor <= max_aceitavel):
        print(f"ALERTA: {nome_sensor} fora do padrao!")
    else:
        print(f"{nome_sensor} OK.")
    print("-" * 20)

# Agora, usamos nossa funcao para cada sensor
verificar_sensor("Umidade do solo", 45.7, "%", 30, 70)
verificar_sensor("Temperatura do ar", 31.5, " C", 22, 30)
verificar_sensor("pH da agua", 5.8, "", 6.5, 8.5)
```

Saída Esperada

```
Lendo dado de Umidade do solo: 45.7\%
Umidade do solo OK.
--------------------
Lendo dado de Temperatura do ar: 31.5°C
ALERTA: Temperatura do ar fora do padrão!
--------------------
Lendo dado de pH da água: 5.8
pH da água OK.
--------------------
```

Observe a clareza do segundo código. A lógica de verificação está definida em um único lugar. Se precisarmos alterar a mensagem de alerta, mudamos apenas dentro da função `verificar_sensor`, e a mudança se aplicará a todos os sensores automaticamente. Isso torna nosso código mais:

- **Legível:** É fácil entender o que o programa faz: ele verifica três sensores.
- **Reutilizável:** Podemos usar a mesma função para um quarto, quinto, ou centésimo sensor sem escrever novo código.
- **Manutenível:** Corrigir um erro ou fazer uma melhoria exige a edição de um único local.

Dominar as funções transforma os algoritmos, tornando-os imensamente mais limpos, organizados e eficientes. Para usá-las corretamente, porém, precisamos entender suas regras e componentes.

2.1.2 Parâmetros e Retorno de Valores: A Comunicação das Funções

No tópico anterior, comparamos funções a equipes de especialistas prontas para realizar uma tarefa. Mas como passamos as instruções corretas para essa equipe? Se temos uma função para calcular a média de notas, como informamos a ela *quais* notas devem ser usadas? E depois que a equipe termina o trabalho, como ela nos entrega o resultado? É aqui que entram dois conceitos fundamentais: **parâmetros** e **retorno de valores**.

- **Parâmetros** são os canais de entrada de uma função. São as informações que enviamos *para dentro* da função para que ela possa trabalhar. Pense neles como os ingredientes que você entrega a um chef de cozinha.
- **Retorno de Valores** é o canal de saída. É o resultado que a função nos devolve *após* ter concluído sua tarefa. É o prato pronto que o chef entrega ao final.

Dominar essa comunicação é essencial para criar funções flexíveis e poderosas, capazes de se adaptar a diferentes dados e cenários. Vamos explorar como essa troca de informações acontece.

A Anatomia de uma Função: Definindo Parâmetros

Quando definimos uma função, declaramos os parâmetros que ela espera receber dentro dos parênteses. Esses parâmetros funcionam como variáveis locais, ou seja, só existem dentro da função. Revisitando o exemplo do monitoramento ambiental, criamos a função `verificar_sensor` que recebia cinco informações, onde cada uma delas era um parâmetro:

Copie e Teste!

```
def verificar_sensor(nome_sensor, valor, unidade, min_aceitavel,
    max_aceitavel):
     # O código da função usa esses parâmetros
     print(f"Lendo dado de {nome_sensor}: {valor}{unidade}")
     # ... resto do código ...
```

Quando **chamamos** a função, os valores que passamos para ela são chamados de **argumentos**. Esses argumentos são atribuídos aos parâmetros na ordem em que são passados.

Copie e Teste!

```
# "Temperatura do ar", 31.5, "°C", 22, 30 são os ARGUMENTOS
verificar_sensor("Temperatura do ar", 31.5, "°C", 22, 30)
```

Nesse caso, dentro da função `verificar_sensor`:

- `nome_sensor` recebe `"Temperatura do ar"`
- `valor` recebe `31.5`
- `unidade` recebe `"°C"`
- `min_aceitavel` recebe `22`
- `max_aceitavel` recebe `30`

Essa forma de passar argumentos, baseada na ordem, é a mais comum e é chamada de **argumentos posicionais**.

Tipos de Argumentos: Flexibilizando a Chamada

Python nos oferece diferentes maneiras de passar argumentos para uma função, o que torna nosso código mais claro e flexível.

1. Argumentos Posicionais

Como vimos, são os argumentos passados na mesma ordem em que os parâmetros foram definidos. A posição de cada argumento determina a qual parâmetro ele será atribuído.

Caso Prático

Você precisa criar uma função que descreva uma espécie de árvore da Amazônia, informando seu nome popular, nome científico e altura média. A ordem das informações é importante.

Copie e Teste!

```
def descrever_especie(nome_popular, nome_cientifico, altura_media)
    :
    """ Descreve uma espécie de árvore com base em seus dados. """
    print(f"--- Ficha da Espécie ---")
    print(f"Nome Popular: {nome_popular}")
    print(f"Nome Científico: {nome_cientifico}")
    print(f"Altura Média: {altura_media} metros")

# Chamando a função com argumentos posicionais
descrever_especie("Sumaúma", "Ceiba pentandra", 40)
```

Saída Esperada

```
--- Ficha da Espécie ---
Nome Popular: Sumaúma
Nome Científico: Ceiba pentandra
Altura Média: 40 metros
```

Fique Alerta!

Com argumentos posicionais, a ordem é crucial! Se você trocar os argumentos de lugar, a função ainda funcionará, mas o resultado estará logicamente incorreto.

```
# Chamada com ordem incorreta
descrever_especie("Ceiba pentandra", 40, "Sumaúma")
```

Saída Incorreta

```
--- Ficha da Espécie ---
Nome Popular: Ceiba pentandra
Nome Científico: 40
Altura Média: Sumaúma metros
```

2. Argumentos Nomeados (Keyword Arguments)

Para evitar a confusão da ordem, Python permite que você especifique explicitamente a qual parâmetro cada argumento pertence. Isso é feito usando o nome do parâmetro seguido de um sinal de igual (=). A grande vantagem é que a ordem dos argumentos nomeados não importa, tornando a chamada da função muito mais legível e segura.

Copie e Teste!

```
def descrever_especie(nome_popular, nome_cientifico, altura_media)
    :
    """ Descreve uma espécie de árvore com base em seus dados. """
    print(f"--- Ficha da Espécie ---")
    print(f"Nome Popular: {nome_popular}")
    print(f"Nome Científico: {nome_cientifico}")
    print(f"Altura Média: {altura_media} metros\n")

# Usando argumentos nomeados, a ordem não importa
print("Chamada 1 (ordem misturada):")
descrever_especie(altura_media=25, nome_popular="Açaí",
    nome_cientifico="Euterpe oleracea")

# Você também pode misturar posicionais e nomeados
# Desde que os posicionais venham primeiro!
print("Chamada 2 (mista):")
descrever_especie("Castanheira", altura_media=50, nome_cientifico=
    "Bertholletia excelsa")
```

Saída Esperada

```
Chamada 1 (ordem misturada):
--- Ficha da Espécie ---
Nome Popular: Açaí
Nome Científico: Euterpe oleracea
Altura Média: 25 metros

Chamada 2 (mista):
--- Ficha da Espécie ---
Nome Popular: Castanheira
Nome Científico: Bertholletia excelsa
Altura Média: 50 metros
```

3. Valores Padrão para Parâmetros

Para evitar ter que digitar a mesma informação repetidamente, podemos definir um **valor padrão**. Isso é feito diretamente na definição da função (`def`), atribuindo um valor ao parâmetro.

Caso Prático

Construa uma função para registrar a medição do nível de um rio. O local padrão é ”Próximo à comunidade ribeirinha”, mas deve ser possível alterá-lo.

Copie e Teste!

```
def registrar_nivel_rio(nivel_metros, local="Próximo à comunidade
    ribeirinha"):
     """ Registra a medição do nível de um rio em um local
    específico. """
     print(f"Medição registrada:")
     print(f"  -> Nível: {nivel_metros} metros")
     print(f"  -> Local: {local}\n")

# Chamada 1: Usando o local padrão
print("--- Medição Padrão ---")
registrar_nivel_rio(3.5)

# Chamada 2: Especificando um local diferente
print("--- Medição em Ponto Específico ---")
registrar_nivel_rio(3.7, local="Foz do igarapé")
```

Saída Esperada

```
--- Medição Padrão ---
Medição registrada:
```

```
  -> Nível: 3.5 metros
  -> Local: Próximo à comunidade ribeirinha

--- Medição em Ponto Específico ---
Medição registrada:
  -> Nível: 3.7 metros
  -> Local: Foz do igarapé
```

Um parâmetro com valor padrão se torna opcional. Se o argumento não for fornecido na chamada, o valor padrão é usado.

Devolvendo Resultados: A Declaração `return`

Até agora, nossas funções realizaram ações, como imprimir informações na tela, mas e se quisermos que a função nos entregue um valor para que possamos usá-lo em outra parte do nosso código? Para tratar exatamente desses casos que utilizamos a declaração `return`.

Quando o Python encontra um `return` dentro de uma função, ele imediatamente sai da função e devolve o valor especificado.

Caso Prático

Uma cooperativa de produtores de castanha-do-pará precisa de uma função para calcular o valor total de um lote com base na quantidade de ”latas”(uma medida comum na região) e no preço por lata. O resultado desse cálculo será usado para gerar uma nota fiscal.

Copie e Teste!

```python
def calcular_valor_lote(quantidade_latas, preco_por_lata):
    """ Calcula o valor total de um lote de castanhas. """
    valor_total = quantidade_latas * preco_por_lata
    return valor_total

# --- Programa Principal ---

# Coletando os dados
latas_vendidas = 50
preco_atual = 35.75 # Preço por lata

# Chamando a função e armazenando o resultado
valor_a_pagar = calcular_valor_lote(latas_vendidas, preco_atual)

# Usando o valor retornado em outra parte do código
print("--- Recibo Simples ---")
print(f"Quantidade: {latas_vendidas} latas")
print(f"Preço por Lata: R$ {preco_atual}")
print(f"Valor Total a Pagar: R$ {valor_a_pagar:.2f}")
```

Saída Esperada

```
--- Recibo Simples ---
Quantidade: 50 latas
Preço por Lata: R$ 35.75
Valor Total a Pagar: R$ 1787.50
```

Perceba a diferença: a função `calcular_valor_lote` não imprime nada. Sua única responsabilidade é fazer o cálculo e **devolver** o resultado. O programa principal é quem decide o que fazer com esse valor (neste caso, imprimi-lo de forma formatada). Isso torna a função muito mais reutilizável.

Retornando Múltiplos Valores

Quando uma função precisa retornar mais de um informação, podemos fazer isso em Python separando os valores com uma vírgula no `return`. Quando você faz isso, a função, na verdade, agrupa os valores em uma **tupla** antes de retornar.

Caso Prático

Crie uma função que receba uma lista de medições de temperatura e retorne a temperatura mínima e a máxima registradas.

Copie e Teste!

```
def analisar_temperaturas(medicoes):
    """ Encontra a temperatura mínima e máxima de uma lista de
   medições. """
    if not medicoes: # Verifica se a lista não está vazia
        return (None, None) # Retorna None se não houver dados

    minima = min(medicoes)
    maxima = max(medicoes)
    return minima, maxima # Retorna os dois valores

# Lista de temperaturas coletadas durante uma semana
temperaturas_semana = [28.5, 29.1, 31.2, 27.8, 30.5, 32.0, 29.9]

# Chamando a função e "desempacotando" o resultado em duas
   variáveis
temp_min, temp_max = analisar_temperaturas(temperaturas_semana)

print(f"Análise da Semana:")
print(f"  - Temperatura Mínima: {temp_min}°C")
print(f"  - Temperatura Máxima: {temp_max}°C")

# O que acontece se chamarmos com uma lista vazia?
min_vazia, max_vazia = analisar_temperaturas([])
print(f"\nAnálise de lista vazia: {min_vazia}, {max_vazia}")
```

Saída Esperada

```
Análise da Semana:
  - Temperatura Mínima: 27.8°C
  - Temperatura Máxima: 32.0°C

Análise de lista vazia: None, None
```

O Retorno Implícito de `None`

O que acontece se uma função não tiver uma declaração `return`? Toda função em Python retorna algo. Se você não especificar o que ela deve retornar, ela retornará implicitamente o valor especial `None`. `None` é a forma do Python de representar "a ausência de valor". É útil para indicar que uma operação não produziu um resultado válido.

Copie e Teste!

```
def saudar(nome):
    """ Uma função que apenas realiza uma ação (imprimir) e não
   retorna nada. """
    print(f"Olá, {nome}! Seja bem-vindo(a) ao projeto CITHA.")

resultado = saudar("Estudante")

print(f"\nO valor retornado pela função saudar é: {resultado}")
print(f"O tipo do valor retornado é: {type(resultado)}")
```

Saída Esperada

```
Olá, Estudante! Seja bem-vindo(a) ao projeto CITHA.

O valor retornado pela função saudar é: None
O tipo do valor retornado é: <class 'NoneType'>
```

Entender parâmetros e retornos é como aprender a gramática da comunicação entre diferentes partes do seu código. Com esse conhecimento, você pode criar "ferramentas"(funções) que recebem os "materiais"corretos (parâmetros) e entregam o "produto"esperado (valor de retorno), tornando seus programas modulares, organizados e muito mais poderosos.

2.1.3 Escopo de Variáveis: Onde as Variáveis Vivem

No tópico anterior, vimos as funções como "equipes de especialistas"que realizam tarefas. Compará-las à construção de uma casa foi útil: temos a equipe da fundação, a dos eletricistas, etc. Agora, vamos aprofundar essa analogia.

Pense em cada função como um cômodo diferente da casa. O que acontece dentro de um cômodo, fica no cômodo. As ferramentas e materiais que você leva para dentro do quarto

para montar um móvel (variáveis) pertencem àquele quarto e, quando você termina a tarefa e sai, aquelas ferramentas são ”guardadas”e não estão mais à vista na sala de estar.

Por outro lado, a estrutura da casa, como as paredes e o teto, é visível de todos os cômodos. Este é o conceito de **escopo**: ele define as ”paredes”que determinam onde uma variável é conhecida e pode ser utilizada. Entender essas regras é crucial para evitar erros e escrever um código limpo e previsível. Vamos explorar os dois principais ”ambientes”onde nossas variáveis podem viver: o escopo local e o escopo global.

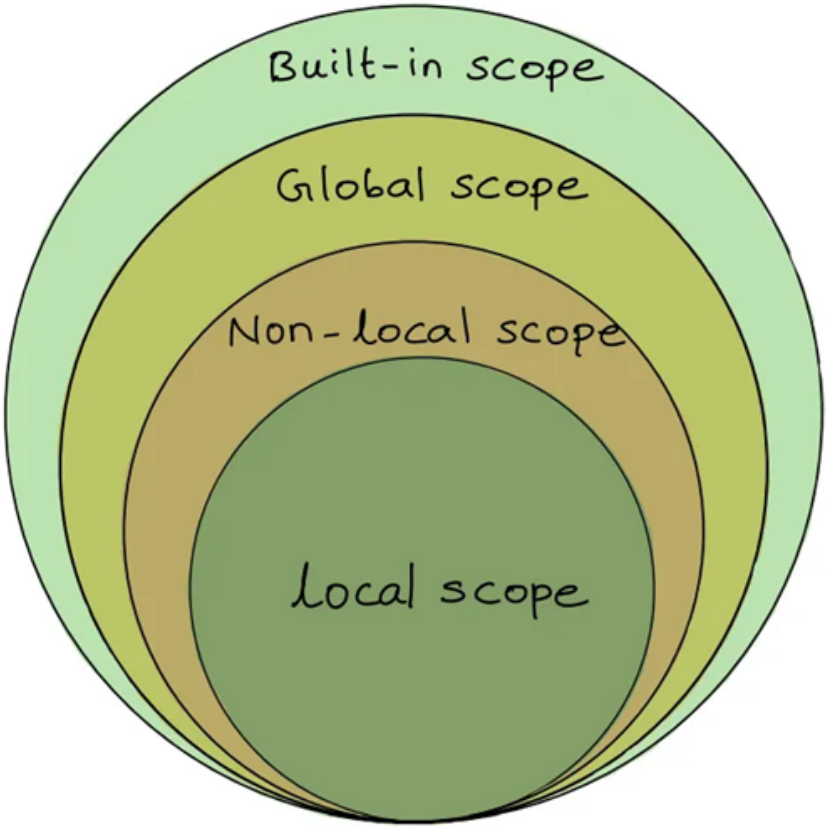

Figura 2.2: Representação da hierarquia do escopo de variáveis em Python
Fonte: Subhadra Bhupathiraju, 2023

O diagrama 2.2 é uma ótima representação a respeito da ordem exata que o Python usa para encontrar uma variável, conhecida como Regra LEGB:

- **L (Local):** Primeiro, Python procura dentro da função atual.
- **E (Enclosing/Não-Local):** Se não encontrar, ele procura no escopo de qualquer função que ”envolva”a função atual (em caso de funções aninhadas).
- **G (Global):** Em seguida, a busca vai para o escopo global do arquivo.
- **B (Built-in):** Por último, ele verifica as funções e nomes que já vêm com o Python, como `print()` e `len()`.

É por esse motivo que uma variável local ”esconde”uma global de mesmo nome, já que o Python a encontra primeiro no círculo mais interno e para a busca ali mesmo!

Escopo Local: O Mundo Privado da Função

Uma variável criada *dentro* de uma função possui um **escopo local**. Isso significa que ela nasce, vive e morre exclusivamente dentro daquela função. Nenhuma outra parte do seu programa, fora daquela função específica, sabe de sua existência.

Essa característica é uma grande vantagem, pois garante que as funções não interfiram umas nas outras. Cada uma tem seu próprio ”espaço de trabalho”isolado, evitando que a variável x de uma função modifique acidentalmente a variável x de outra.

Caso Prático

Uma equipe de biólogos está monitorando uma área de reflorestamento e precisa de uma função simples para calcular a densidade de árvores em uma parcela. A função receberá o número de árvores contadas e a área da parcela em hectares. O cálculo da densidade será uma variável local.

Copie e Teste!

```
def calcular_densidade_arvores(contagem_arvores, area_hectares):
    """Calcula a densidade de árvores por hectare."""
    # 'densidade' é uma variável de escopo local.
    # Ela só existe dentro desta função.
    densidade = contagem_arvores / area_hectares
    print(f"-> Dentro da função: A densidade calculada é de {
   densidade:.2f} árvores/ha.")
    return densidade

# --- Programa Principal ---
print("Iniciando a análise da Parcela 1...")
densidade_parcela1 = calcular_densidade_arvores(520, 2) # 520
   árvores em 2 hectares
print(f"Fora da função: O resultado retornado foi {
   densidade_parcela1:.2f}.\n")

# Agora, vamos tentar acessar a variável 'densidade' diretamente.
print("Tentando acessar a variável 'densidade' fora da função...")
print(densidade)
```

Saída Esperada

```
Iniciando a análise da Parcela 1...
-> Dentro da função: A densidade calculada é de 260.00
   árvores/ha.
Fora da função: O resultado retornado foi 260.00.

Tentando acessar a variável 'densidade' fora da função...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'densidade' is not defined
```

Ao executar este código, você notará algo importante. A primeira parte funciona perfeitamente, mas a última linha gera um erro.

Fique Alerta!

O `NameError` é a forma do Python dizer: ”Procurei pela variável chamada `densidade` aqui fora, no programa principal, e não a encontrei!”. Isso acontece porque, assim que a função `calcular_densidade_arvores` terminou seu trabalho e retornou o resultado, a variável local `densidade` foi destruída. Ela cumpriu seu propósito e deixou de existir. É por isso que usamos o `return`: para trazer um valor de dentro do ”cômodo”(escopo local) para o ”corredor principal”(escopo global).

Escopo Global: Variáveis Visíveis para Todos

Em contrapartida ao escopo local, uma variável definida no corpo principal do seu arquivo, fora de qualquer função, possui um **escopo global**. Retornando à nossa analogia da casa, uma variável global é como a rede Wi-Fi: seu sinal está disponível em todos os cômodos. Qualquer função no seu programa pode *ler* ou *acessar* o valor de uma variável global.

Caso Prático

Em muitos estudos ambientais, é preciso converter a biomassa de uma floresta em estoque de carbono. O fator de conversão é geralmente um valor constante (aproximadamente 0.5, significando que 50% da biomassa é carbono). Podemos definir este fator como uma variável global, para que diferentes funções de cálculo possam usá-lo sem a necessidade de passá-lo como parâmetro todas as vezes.

Copie e Teste!

```
# FATOR_CARBONO é uma variável de escopo global.
# Por convenção, nomes em maiúsculas indicam constantes.
FATOR_CARBONO = 0.5

def estimar_carbono_floresta(biomassa_total_toneladas):
    """Estima o carbono estocado com base na biomassa, usando o
   fator global."""
    print("-> Acessando a variável global FATOR_CARBONO de dentro
   da função.")
    # A função PODE LER a variável global sem problemas.
    carbono_estocado = biomassa_total_toneladas * FATOR_CARBONO
    return carbono_estocado

def estimar_carbono_solo(materia_organica_toneladas):
    """Estima o carbono no solo, que também usa o mesmo fator
   global."""
    print("-> Outra função acessando a mesma variável global.")
    carbono_no_solo = materia_organica_toneladas * FATOR_CARBONO
    return carbono_no_solo
```

```
# --- Programa Principal ---
estoque_carbono_floresta = estimar_carbono_floresta(1200)
print(f"Estoque de carbono na floresta: {estoque_carbono_floresta}
    toneladas.\n")

estoque_carbono_solo = estimar_carbono_solo(350)
print(f"Estoque de carbono no solo: {estoque_carbono_solo}
   toneladas.")
```

Saída Esperada

```
-> Acessando a variável global FATOR_CARBONO de dentro da
   função.
Estoque de carbono na floresta: 600.0 toneladas.

-> Outra função acessando a mesma variável global.
Estoque de carbono no solo: 175.0 toneladas.
```

O Cuidado ao Tentar Modificar Variáveis Globais

Até aqui, tudo parece simples: as funções podem ler variáveis globais. Mas o que acontece se uma função tentar *alterar* ou *modificar* o valor de uma variável global?

Por padrão, Python protege o escopo global. Se você tentar atribuir um novo valor a uma variável dentro de uma função que tem o mesmo nome de uma variável global, o Python, em vez de alterar a global, criará uma **nova variável local** com aquele nome. A variável global original permanecerá intacta. Este comportamento, conhecido como ”sombreamento”(*shadowing*), pode causar muita confusão.

Copie e Teste!

```
contador_alertas = 0 # Global

def registrar_alerta(tipo_alerta):
    print(f"\n ALERTA REGISTRADO: {tipo_alerta}")
    # Aqui, Python cria uma NOVA variável 'contador_alertas' que é
    LOCAL
    contador_alertas = 1
    print(f"-> Contagem local de alertas nesta função: {
   contador_alertas}")

print(f"Contagem total de alertas ANTES da chamada: {
   contador_alertas}")
registrar_alerta("Nível do rio muito alto")
print(f"Contagem total de alertas DEPOIS da chamada: {
   contador_alertas}")
```

Saída Esperada

```
Contagem total de alertas ANTES da chamada: 0

ALERTA REGISTRADO: Nível do rio muito alto
-> Contagem local de alertas nesta função: 1
Contagem total de alertas DEPOIS da chamada: 0
```

Note que o contador global não foi alterado! A função criou sua própria variável local e a modificou, deixando a global intacta.

A Palavra-Chave `global`

Como, então, podemos *realmente* modificar uma variável global de dentro de uma função? Para isso, usamos a palavra-chave **global**. Ela avisa ao Python: "Ei, nesta função, quando eu me referir a esta variável, estou falando daquela que existe no escopo global, não crie uma nova."

Copie e Teste!

```python
contador_alertas = 0 # Global

def registrar_alerta_global(tipo_alerta):
    global contador_alertas # Avisando ao Python: use a variável
    global!
    print(f"\n ALERTA REGISTRADO: {tipo_alerta}")
    contador_alertas = contador_alertas + 1
    print(f"-> Contagem de alertas atualizada: {contador_alertas}"
    )

print(f"Contagem inicial: {contador_alertas}")
registrar_alerta_global("Temperatura elevada")
registrar_alerta_global("Desmatamento detectado")
print(f"Contagem final: {contador_alertas}")
```

Saída Esperada

```
Contagem inicial: 0

ALERTA REGISTRADO: Temperatura elevada
-> Contagem de alertas atualizada: 1

ALERTA REGISTRADO: Desmatamento detectado
-> Contagem de alertas atualizada: 2
Contagem final: 2
```

Fique Alerta!

Use a palavra-chave `global` com **extremo cuidado**. Modificar variáveis globais de dentro de muitas funções pode tornar seu código muito difícil de entender e depurar. Se uma variável global muda de valor, você terá que investigar todas as funções que a utilizam para descobrir qual delas fez a alteração.

Em geral, a melhor prática é preferir a comunicação via **parâmetros e `return`**. Se uma função precisa de um valor, passe-o como parâmetro. Se ela produz um resultado, retorne-o. Esta abordagem, discutida em textos clássicos como Lutz (2013), mantém suas funções independentes e seu código mais organizado e previsível.

2.1.4 Boas Práticas: Docstrings e Sugestões de Tipo (Type Hints)

Até agora, você aprendeu a construir funções, que são as ”equipes de especialistas”do nosso código. Mas como garantimos que outra pessoa (ou até mesmo você, daqui a alguns meses) entenda exatamente o que sua equipe faz, quais ”ferramentas”ela precisa e o que ela ”entrega”no final? E como podemos deixar essas instruções tão claras que até o próprio editor de código nos ajude a evitar erros?

Nesta seção, vamos refinar nossas habilidades e aprender a escrever funções como um profissional. Abordaremos duas práticas essenciais que, embora não alterem o que a função faz, transformam radicalmente a clareza, a segurança e a manutenibilidade do nosso código: as **Docstrings** e as **Sugestões de Tipo (Type Hints)**.

Docstrings: O Manual de Instruções da sua Função

Imagine que você criou uma ferramenta incrível, como um dispositivo que mede a pureza da água de um rio. Se você entregar essa ferramenta a um colega sem nenhum manual, ele provavelmente não saberá como usá-la, para que serve cada botão ou como interpretar os resultados. O mesmo acontece com nossas funções.

Uma **docstring** (documentation string, ou ”string de documentação”) é o manual de instruções oficial de uma função em Python. É um texto, delimitado por aspas triplas (`"` ou `'''`), que fica na primeira linha logo após a definição da função. Seu propósito é explicar, de forma clara e concisa, o que a função faz.

Fique Alerta!

Uma docstring não é um comentário (#). Comentários são feitos para explicar *como* uma parte específica do código funciona. Docstrings explicam *o quê* a função como um todo faz, para quem vai usá-la sem precisar ler todo o seu código interno. Além disso, ferramentas automáticas e a função nativa `help()` utilizam as docstrings.

Embora você possa escrever qualquer coisa em uma docstring, a comunidade Python adota padrões para que elas sejam fáceis de ler e processar por ferramentas. Um formato comum e legível inclui:

1. **Resumo de uma linha:** O que a função faz, de forma direta.
2. **Descrição detalhada (opcional):** Mais detalhes sobre o comportamento da função.

3. **Parâmetros (Argumentos):** Uma lista de cada parâmetro, seu tipo esperado e o que ele representa.
4. **Retorno:** O que a função retorna, seu tipo e o que esse valor significa.

Caso Prático

Uma ONG de conservação precisa de uma função para estimar o estoque de carbono de uma árvore, uma métrica vital para projetos de reflorestamento na Amazônia. A função deve receber a espécie da árvore e o diâmetro do seu tronco em centímetros, e retornar uma estimativa da biomassa em quilogramas. Vamos criar essa função e documentá-la corretamente.

Copie e Teste!

```
def estimar_biomassa_arvore(especie, diametro_cm):
    """Estima a biomassa de uma árvore com base em sua espécie e
   diâmetro. Utiliza um fator de conversão simplificado que varia
   conforme a espécie para calcular a biomassa acima do solo. Não
   é um modelo científico preciso, mas serve como exemplo.

    Args:
        especie (str): O nome popular da espécie da árvore.
        diametro_cm (float): O diâmetro do tronco medido à altura
   do peito (DAP), em centímetros.

    Returns:
        float: A biomassa estimada da árvore em quilogramas (kg)."
   ""
    if especie == "Sumaúma":
        fator = 0.65
    else:
        fator = 0.55  # Fator genérico para outras espécies

    # Fórmula simplificada: biomassa = fator * (diametro^2)
    biomassa_kg = fator * (diametro_cm ** 2)
    return biomassa_kg

# Agora, vamos ver como acessar nossa documentação
help(estimar_biomassa_arvore)
```

Saída Esperada

```
Help on function estimar_biomassa_arvore in module __main__:

estimar_biomassa_arvore(especie, diametro_cm)
    Estima a biomassa de uma árvore com base em sua espécie
   e diâmetro. Esta função utiliza um fator de conversão
   simplificado que varia conforme a espécie para calcular a
    biomassa acima do solo. Não é um modelo científico
```

```
    preciso, mas serve como exemplo.

     Args:
         especie (str): O nome popular da espécie da árvore.
         diametro_cm (float): O diâmetro do tronco medido à
    altura do peito (DAP), em centímetros.

     Returns:
         float: A biomassa estimada da árvore em quilogramas
    (kg)
```

Type Hints: Deixando seu Código Mais Claro e Seguro

Se as docstrings são o manual de instruções, as **sugestões de tipo (type hints)** são como uma lista de ingredientes clara e específica. Em vez de dizer "adicione açúcar", a receita diz "adicione 1 xícara de açúcar refinado". Essa precisão extra ajuda a evitar erros e torna o processo muito mais claro.

Type hints são uma forma moderna, introduzida nas versões mais recentes do Python, de indicar os tipos de dados esperados para os parâmetros de uma função e para o valor que ela retorna. A sintaxe é simples:

- `parametro: tipo` para parâmetros.
- `-> tipo_retorno` para o valor de retorno, colocado antes dos dois-pontos da função.

Vamos aprimorar nossa função `estimar_biomassa_arvore` com type hints.

Copie e Teste!

```
def estimar_biomassa_arvore_com_tipos(especie: str, diametro_cm:
    float) -> float:
     """Estima a biomassa de uma árvore com base em sua espécie e
    diâmetro.

     Args:
         especie (str): O nome popular da espécie da árvore
         diametro_cm (float): O diâmetro do tronco medido à altura
    do peito (DAP), em centímetros.

     Returns:
         float: A biomassa estimada da árvore em quilogramas (kg)""
    "
     if especie == "Sumaúma":
         fator = 0.65
     else:
         fator = 0.55

     biomassa_kg = fator * (diametro_cm ** 2)
     return biomassa_kg
```

```python
# Chamada correta
biomassa_castanheira = estimar_biomassa_arvore_com_tipos("
    Castanheira", 120.5)
print(f"Biomassa da Castanheira: {biomassa_castanheira:.2f} kg")

# O que acontece se passarmos os tipos errados?
# O código AINDA RODA, mas um verificador de tipos acusaria um
    erro aqui!
biomassa_errada = estimar_biomassa_arvore_com_tipos(150.0, "Açaí")
print(f"Resultado da chamada incorreta: {biomassa_errada}")
```

Saída Esperada

```
Biomassa da Castanheira: 7986.14 kg
Traceback (most recent call last):
  File "<python-input-0>", line 24, in <module>
    biomassa_errada = estimar_biomassa_arvore_com_tipos
   (150.0, "Açaí")

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^
  File "<python-input-0>", line 15, in
   estimar_biomassa_arvore_com_tipos
    biomassa_kg = fator * (diametro_cm ** 2)
                           ~~~~~~~~~~~~~^^~~
TypeError: unsupported operand type(s) for ** or pow(): 'str
    ' and 'int'
```

Podemos verificar que o primeiro caso rodou corretamente, mas o segundo caso ele acusa um erro na função `biomassa_errada`. Então ele descreve que essa linha utiliza a função `biomassa_kg` a qual não está conseguindo fazer a operação `diametro_cm ** 2`, através da marcação `~~~~^^~~`. Analisando o TypeError, verificamos que a falha está na operação de potencia (** indicada pela marcação ^^), afinal estamos tentando elevar ao quadrado uma string.

Saída Esperada

```
teste_tipos.py:9: error: Argument 1 to "
    estimar_biomassa_arvore" has incompatible type "float";
    expected "str"  [arg-type]
teste_tipos.py:9: error: Argument 2 to "
    estimar_biomassa_arvore" has incompatible type "str";
    expected "float"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

Verificando o Type Hint, é possível perceber que há incompatibilidade relacionada ao tipo das variáveis em ambos os argumentos da função `estimar_biomassa_arvore`.

Os benefícios dos type hints são imensos, especialmente em projetos maiores, conforme ressaltado por autores como Luciano Ramalho em sua obra "Python Fluente"(Ramalho, 2015):

- **Legibilidade:** Fica instantaneamente claro que tipo de dado a função espera e retorna.
- **Detecção de Erros:** Editores de código modernos (IDEs) e ferramentas externas como o `mypy` usam esses "hints"para verificar seu código *antes* de você executá-lo, alertando sobre possíveis erros de tipo.
- **Autocompletar Inteligente:** O editor de código consegue identificar que a variável `biomassa_castanheira` é um `float`, então ele irá sugerir métodos aplicáveis a números, e não a strings.
- **Facilita a Colaboração:** Quando outra pessoa for usar sua função, ela saberá exatamente que tipo de dado deve passar.

Tipos Mais Complexos E se sua função trabalha com listas ou dicionários? O módulo `typing` do Python nos dá acesso a tipos mais específicos.

Caso Prático

Um sistema de monitoramento ambiental coleta dados de vários sensores e os armazena em uma lista de dicionários. Cada dicionário contém o nome do sensor e seu último valor medido. Precisamos de uma função que processe essa lista e retorne o nome do sensor com a maior leitura.

Copie e Teste!

```
# Importamos os tipos que precisamos do módulo typing
from typing import List, Dict, Optional

def encontrar_sensor_com_maior_valor(leituras: List[Dict[str,
    float]]) -> Optional[str]:
    """Analisa uma lista de leituras de sensores e encontra o
    sensor com maior valor.

    Args:
        leituras: Uma lista onde cada item é um dicionário.
                  Cada dicionário deve ter uma chave 'sensor' (str
    ) e 'valor' (float).

    Returns:
        O nome (str) do sensor que registrou o maior valor.
        Retorna None se a lista de leituras estiver vazia.
    """
    if not leituras:
        return None  # Retorna None para indicar que nenhum sensor
    foi encontrado

    maior_leitura = -1.0
    sensor_destaque = ""
```

```
    for leitura in leituras:
        # Usamos .get() para acessar as chaves de forma segura
        nome_sensor = leitura.get('sensor', 'Desconhecido')
        valor = leitura.get('valor', -1.0)

        if valor > maior_leitura:
            maior_leitura = valor
            sensor_destaque = nome_sensor

    return sensor_destaque

# Dados coletados de um igarapé
dados_sensores = [
    {'sensor': 'pH', 'valor': 6.8},
    {'sensor': 'Turbidez', 'valor': 4.5},
    {'sensor': 'Oxigênio Dissolvido', 'valor': 7.2}
]

sensor_alerta = encontrar_sensor_com_maior_valor(dados_sensores)
print(f"Sensor com maior leitura: {sensor_alerta}")

# Testando com uma lista vazia
sensor_vazio = encontrar_sensor_com_maior_valor([])
print(f"Resultado para lista vazia: {sensor_vazio}")
```

Saída Esperada

```
Sensor com maior leitura: Oxigênio Dissolvido
Resultado para lista vazia: None
```

Neste exemplo, `List[Dict[str, float]]` informa que esperamos uma lista do tipo (`List`) onde cada elemento é um dicionário (`Dict`) com chaves do tipo string (`str`) e valores do tipo float (`float`). `Optional[str]` no retorno indica que a função pode retornar uma string ou `None`, tornando o código mais seguro e explícito sobre seus possíveis resultados. A adoção dessas práticas segundo Lutz (2013), é um passo fundamental para escrever código robusto e de fácil manutenção.

2.1.5 Funções Anônimas (Lambda)

Até agora, todas as funções que criamos tinham um nome, como `calcular_media` ou `verificar_sensor`. Nós as definimos com `def` porque planejávamos reutilizá-las em vários pontos do nosso programa. Mas e se precisarmos de uma função muito simples, para uma tarefa rápida e que só será usada uma única vez? Seria como construir uma ferramenta complexa e permanente para apertar um único parafuso. Para essas situações o Python nos oferece uma sintaxe mais direta e compacta, as **funções anônimas** mais conhecidas como **funções lambda**.

Pense em uma função lambda como um ”ajudante temporário”. Em vez de contratar um especialista em tempo integral (uma função `def`), você chama um assistente para uma tarefa

única e imediata. Essas funções não têm nome ("anônimas") e são definidas em uma única linha de código, o que as torna perfeitas para operações pequenas e pontuais.

A Sintaxe de uma Função Lambda

A estrutura de uma função lambda é notavelmente simples e foi projetada para ser concisa, se resumindo a uma única linha:

```
lambda argumentos: expressao
```

- `lambda`: É a palavra-chave que sinaliza ao Python a criação de uma função anônima.
- `argumentos`: São os parâmetros que a função recebe, exatamente como em uma função `def`, mas sem a necessidade de parênteses. Se houver mais de um, eles são separados por vírgulas (ex: `lambda x, y:`).
- `:`: O caractere de dois-pontos separa os argumentos da operação que será realizada.
- `expressao`: Esta é a parte crucial, uma **única expressão** (uma operação, um cálculo) que a função executa. O resultado desta expressão é automaticamente retornado, sem a necessidade da palavra-chave `return`.

Caso Prático

Em seu projeto de monitoramento na Amazônia, você frequentemente precisa converter o diâmetro de uma árvore para o seu raio. Implemente uma solução que compare o paradigma tradicional (def) com o paradigma funcional (lambda).

Copie e Teste!

```
# Usando uma função def tradicional
def calcular_raio(diametro):
    return diametro / 2

# A mesma lógica com uma função lambda
calcular_raio_lambda = lambda diametro: diametro / 2

# Vamos testar as duas
diametro_arvore = 90 # em cm
print(f"Raio (calculado com def): {calcular_raio(diametro_arvore)}
    cm")
print(f"Raio (calculado com lambda): {calcular_raio_lambda(
   diametro_arvore)} cm")
```

Saída Esperada

```
Raio (calculado com def): 45.0 cm
Raio (calculado com lambda): 45.0 cm
```

Talvez possa parecer que o fato de atribuímos a função lambda a uma certa variável (`calcular_raio_lambda`), faria com que ela deixasse de ser ”anônima”. Tecnicamente sim, no entanto, este não é o uso mais comum ou poderoso das lambdas. A verdadeira força delas aparece quando as usamos como argumentos para outras funções de ordem superior.

Onde as Lambdas Brilham: Funções como Argumentos

O cenário mais comum para o uso de funções lambda é quando uma função precisa receber outra função como parâmetro. Funções como `sort()`, `map()` e `filter()` em Python operam sobre coleções de dados e, muitas vezes, precisam de uma pequena ”instrução”ou ”regra”para saber *como* devem realizar sua tarefa.

Caso Prático

Uma equipe de pesquisadores coletou dados sobre a turbidez da água em diferentes pontos de um rio. Os dados estão em uma lista de dicionários e a equipe precisa ordenar essa lista, não em ordem alfabética pelo nome do local, mas sim pelo valor da turbidez, do menor para o maior.

Copie e Teste!

```
dados_coleta = [
    {'local': 'Foz do Igarapé', 'turbidez': 4.8},
    {'local': 'Comunidade Ribeirinha', 'turbidez': 2.1},
    {'local': 'Reserva Florestal', 'turbidez': 1.5},
    {'local': 'Ponto de Descarte', 'turbidez': 7.9}
]

# Usando lambda como a "chave" de ordenação para o método sort()
dados_coleta.sort(key=lambda item: item['turbidez'])

print("Dados ordenados por turbidez da água:")
for ponto in dados_coleta:
    print(ponto)
```

Saída Esperada

```
Dados ordenados por turbidez da água:
{'local': 'Reserva Florestal', 'turbidez': 1.5}
{'local': 'Comunidade Ribeirinha', 'turbidez': 2.1}
{'local': 'Foz do Igarapé', 'turbidez': 4.8}
{'local': 'Ponto de Descarte', 'turbidez': 7.9}
```

No exemplo acima, o parâmetro `key` do método `sort()` espera uma função. Essa função receberá cada elemento da lista (cada dicionário, que chamamos de `item`) e deverá retornar o valor que será usado para a ordenação. A função utilizada nesse exemplo tem exatamente esse papel, já que de forma anônima e em uma única linha, `lambda item: item['turbidez']` instrui o `sort` a olhar para o valor da chave `'turbidez'` em cada

dicionário. Criar uma função `def` completa apenas para essa pequena tarefa seria desnecessário.

Limitações e Boas Práticas

Embora úteis, as funções lambda têm uma limitação fundamental: elas só podem conter **uma única expressão**. Você não pode incluir múltiplas linhas de código, laços `for` ou `while`, ou mesmo condicionais complexos (como `if/elif/else`).

Fique Alerta!

Quando usar `lambda` e quando usar `def`?
A regra de ouro é a **legibilidade**. Use uma função lambda para tarefas simples e curtas, especialmente quando passadas como argumento. Se a lógica começar a ficar complexa ou difícil de ler em uma única linha, não hesite em usar uma função `def` tradicional.

Como aponta (Lutz, 2013), um código claro e explícito é quase sempre preferível a um código compacto, mas difícil de entender. As funções lambda são uma ferramenta elegante no arsenal de um programador Python, promovendo um código mais conciso para tarefas específicas. Dominá-las é mais um passo em direção à escrita de um código mais eficiente e idiomático.

Conhecendo um pouco mais!

Material Extra
Para explorar mais exemplos e casos de uso das funções lambda, a documentação oficial do Python e tutoriais online são excelentes recursos.

`https://docs.python.org/pt-br/3/tutorial/controlflow.html#lambda-expressions`

2.2 Manipulação de Arquivos

No início deste capítulo, comparamos a criação de um programa à construção de uma casa, onde as funções são as equipes especializadas que realizam tarefas específicas, como a instalação elétrica ou a fundação. Essa abordagem modular nos permite construir códigos complexos de forma organizada e reutilizável. Contudo, de que adiantaria construir uma casa perfeitamente funcional se, ao final do dia, tudo o que foi colocado dentro dela simplesmente desaparecesse?

As variáveis e estruturas de dados que utilizamos até agora, como listas e dicionários, vivem na memória volátil do computador (RAM). Isso significa que, assim que o programa termina sua execução, toda a informação coletada, processada ou gerada é perdida para sempre. É como se a nossa casa fosse reiniciada ao estado original cada vez que saímos dela. Para que nossos programas sejam verdadeiramente úteis, eles precisam de memória. Precisam da capacidade de salvar informações de forma permanente, para que possam ser recuperadas e utilizadas mais tarde. Essa capacidade é chamada de **persistência de dados**.

É aqui que entra a manipulação de arquivos. Arquivos são a principal maneira pela qual um programa interage com o armazenamento de longo prazo de um computador, como o disco rígido (HD) ou o SSD. Aprender a ler e escrever em arquivos é o que permite que nossos progra-

mas ”lembrem”de resultados, ”carreguem”configurações salvas e processem grandes volumes de informação que não caberiam na memória de uma só vez. Dominar essa habilidade é um passo fundamental que transforma scripts simples, que executam e esquecem, em aplicações robustas e com memória.

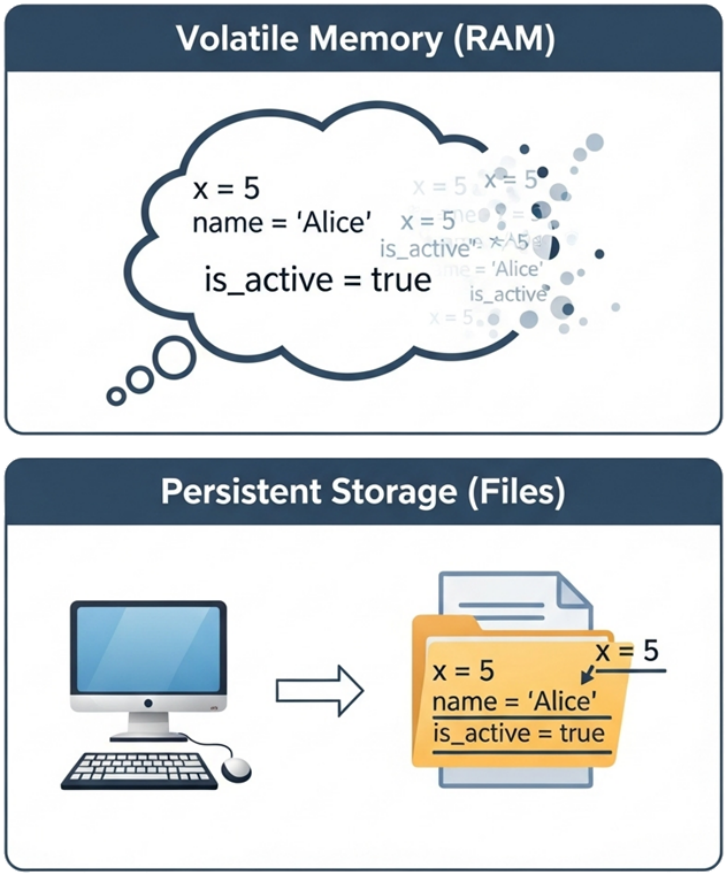

Figura 2.3: Ilustração de conceito de persistencia de dados
Fonte: Gerada por Google Gemini 2.5 Pro, 2025

Nesta seção, vamos abrir as portas e janelas dos nossos programas para o mundo exterior. Começaremos entendendo o básico: o que são arquivos e como os computadores os localizam através de caminhos (*paths*). Em seguida, colocaremos a mão na massa, aprendendo a ler e a escrever nos tipos de arquivos mais comuns:

- **Arquivos de Texto (`.txt`):** A forma mais simples de persistência, ideal para salvar anotações, registros de eventos (*logs*) ou resultados textuais de forma direta e legível por humanos.
- **Arquivos CSV (`Comma-Separated Values`):** A espinha dorsal para dados tabulares. Aprenderemos a manipular esse formato, que é universalmente usado para exportar dados de planilhas como Excel e Google Sheets, sendo essencial para qualquer tarefa que envolva análise de dados.
- **Arquivos JSON (`JavaScript Object Notation`):** O padrão moderno para a troca de dados estruturados na web. Veremos como trabalhar com JSON, um formato leve e legível que é fundamental para interagir com APIs e sistemas online, desde redes sociais até bancos de dados complexos.

Ao final desta seção, você estará apto a criar programas que não apenas executam uma lógica interna, mas que também dialogam com o sistema de arquivos, salvando seu progresso e interagindo com o vasto universo de dados que existe fora do seu código. Como ressaltado por autores clássicos da área, a interação com arquivos é uma das competências essenciais que marcam a transição para a criação de software completo e funcional (Lutz, 2013).

2.2.1 Entendendo Arquivos e Caminhos

Até este ponto, nossos programas têm sido como uma conversa que se perde no ar. As variáveis que criamos, os cálculos que fazemos e os resultados que geramos existem apenas na memória volátil (RAM) do computador. Quando o programa termina, tudo é esquecido. Mas e se quiséssemos guardar os resultados de um monitoramento ambiental, salvar uma lista de espécies catalogadas na Amazônia ou registrar os dados de um sensor por vários dias?

Para que nossos programas tenham ”memória”, eles precisam aprender a interagir com o sistema de armazenamento de longo prazo do computador (como o HD ou SSD). É aqui que entra a **manipulação de arquivos**. Antes de podermos ler ou escrever informações, no entanto, precisamos entender duas ideias fundamentais que funcionam como o sistema de endereço do mundo digital: o que são **arquivos** e como os computadores os encontram usando **caminhos** (*paths*).

O que é um Arquivo?

Pense no sistema de arquivos do seu computador como um gigantesco armário de arquivamento. Dentro deste armário, há gavetas (diretórios ou pastas), e dentro de cada gaveta, há pastas de papel (os arquivos). Cada arquivo guarda um tipo específico de informação: um pode ser um documento de texto (`.txt`), outro uma foto (`.jpg`) e um terceiro uma planilha de dados (`.csv`).

Para o sistema operacional, um arquivo é simplesmente um recurso nomeado que armazena dados de forma permanente. A principal tarefa do nosso programa é saber como pedir ao sistema operacional para abrir a gaveta certa, encontrar a pasta de papel correta e, então, ler ou escrever informações nela.

O Endereço de um Arquivo: O que é um Caminho?

Para encontrar um arquivo nesse imenso armário, não basta saber seu nome; precisamos saber exatamente *onde* ele está guardado. O endereço completo de um arquivo no sistema é chamado de **caminho** (do inglês, *path*). Assim como o endereço da sua casa, um caminho guia o sistema operacional, passo a passo, desde o ponto de partida até o destino final.

Existem duas maneiras de escrever o endereço de um arquivo: usando um caminho absoluto ou um caminho relativo. Antes de explorá-las, precisamos conhecer nosso ponto de partida..

O **diretório de trabalho atual** (em inglês, *Current Working Directory* ou CWD) é a pasta em que seu script Python está sendo executado no momento. Pense nele como o cômodo da casa em que você está agora. Se você está na cozinha, é muito fácil encontrar a ”geladeira”(um arquivo), pois ela está no mesmo cômodo. Mas para encontrar a ”cama”, você precisa saber o caminho para chegar até o quarto.

Saber onde estamos é o primeiro passo para navegar pelo sistema de arquivos. Em Python, podemos descobrir nosso diretório de trabalho atual usando a biblioteca `os` (Operating System).

Copie e Teste!

```python
import os

# Retorna e imprime o diretório de trabalho atual
diretorio_atual = os.getcwd()

print("Meu programa está rodando no seguinte diretório:")
print(diretorio_atual)
```

Saída Esperada

```
Meu programa está rodando no seguinte diretório:
C:\Users\Aluno\Documentos\ProjetoCITHA
```

Tipos de Caminhos: Absoluto vs. Relativo

Agora que sabemos nosso ponto de partida com o CWD, vamos entender as duas formas de escrever o "endereço"de um arquivo. Para visualizar, imagine a estrutura de um projeto de pesquisa guardado no computador:

Saída Esperada

```
/ProjetoCITHA/
|
|-- □ dados/
|    |
|    `-- □ castanheiras.csv
|
`-- □ scripts/
     |
     `-- □ analise.py  <== Você está aqui (CWD)
```

Para chegar em `castanheiras.csv` a partir de `analise.py`:

- **Caminho Relativo:** `../dados/castanheiras.csv`
  - `..` (sobe para a pasta pai: `/ProjetoCITHA/`)
  - `dados/` (entra no diretório `dados`)
  - `castanheiras.csv` (acessa o arquivo)

- **Caminho Absoluto (Ex. Linux/macOS):**
  `/home/aluno/ProjetoCITHA/dados/castanheiras.csv`

- **Caminho Absoluto (Ex. Windows):**
  `C:\Users\Aluno\ProjetoCITHA\dados\castanheiras.csv`

Neste projeto, nosso script `analise.py` (nosso CWD) precisa ler os dados do arquivo `castanheiras.csv`. Vejamos as duas maneiras de informar ao script onde encontrar este arquivo.

1. Caminho Absoluto: O Endereço Completo

Um **caminho absoluto** é o endereço completo de um arquivo ou diretório, começando do ponto mais alto da hierarquia: o diretório raiz. É como dar um endereço postal completo, incluindo país, estado, cidade e rua. Não importa onde você esteja (qual seja seu CWD), um caminho absoluto sempre levará ao mesmo lugar. A aparência do diretório raiz muda conforme o sistema operacional:

- **Windows:** Começa com a letra de uma unidade, como `C:\`.
- **Linux e macOS:** Começa com uma barra `/`.

Caso Prático

Imagine que o projeto CITHA possui uma estação meteorológica autônoma na Reserva Ducke, que salva os dados de chuva em uma pasta central e segura no servidor do IFAM. Vários scripts de análise, localizados em pastas de projetos diferentes, precisam acessar esses dados confiavelmente. Para isso, eles usam o caminho absoluto, que funciona como um endereço fixo, independente da localização do script.

```
C:\DadosCITHA\Sensores\Pluviometro\chuvas_julho.txt
```

Analisando o caminho, cada parte nos guia precisamente até o destino final:
- `C:\` nos posiciona na raiz do sistema de arquivos do servidor.
- `DadosCITHA\Sensores\Pluviometro\` navega até o diretório correto.
- `chuvas_julho.txt` é o nosso arquivo de dados alvo.

Um caminho absoluto é útil quando a localização de um recurso importante é fixa e seu programa precisa encontrá-lo independentemente de onde o script foi executado. Essa é uma prática fundamental para a automação de análises científicas, onde a robustez do código é essencial.

2. Caminho Relativo: Instruções a Partir de Onde Você Está

Um **caminho relativo** é o endereço de um arquivo ou diretório baseado na sua localização atual (o CWD). É como dar instruções a alguém que já está no mesmo bairro: ”siga por esta rua e vire na segunda à direita”. Essas instruções só fazem sentido a partir daquele ponto de partida específico.

Caminhos relativos são extremamente úteis porque tornam seu projeto portátil. Se você mover toda a pasta do seu projeto para outro computador, os caminhos relativos dentro dele continuarão funcionando, pois a relação entre os arquivos e as subpastas não mudou. Utilizamos duas notações especiais em caminhos relativos:

- `.` (um ponto): representa o diretório atual.
- `..` (dois pontos): representa o diretório ”pai”(o diretório um nível acima do atual).

Caso Prático

Sua equipe de desenvolvimento está trabalhando em um projeto de análise de dados de desmatamento. A estrutura do projeto é a seguinte:

```
ProjetoDesmatamento/
|-- scripts/
|   |-- analisar_dados.py  <-- Nosso diretório de trabalho (CWD)
|-- dados/
|   |-- imagens_satelite/
|   |   |-- regiao_norte_2024.jpg
|   |-- relatorios/
|       |-- consolidado_2023.csv
```

Se nosso CWD é `ProjetoDesmatamento/scripts/`, aqui estão alguns caminhos relativos:

- `../dados/relatorios/consolidado_2023.csv` para acessar o arquivo `consolidado_2023.csv`. Isso se traduz em: "suba um nível (`..`), depois entre na pasta `dados`, depois na `relatorios`, e lá está o arquivo".
- `../dados/imagens_satelite/regiao_norte_2024.jpg` para acessar o arquivo `regiao_norte_2024.jpg`.

Construindo Caminhos da Maneira Certa: `os.path.join()`

Você pode ser tentado a criar caminhos juntando strings com +, como `"dados"+ "\\"+ "relatorio.csv"`. **Não faça isso!** O separador de diretórios é diferente entre sistemas operacionais (\ no Windows e / no Linux/macOS).

A maneira correta e segura de construir caminhos é usar a função `os.path.join()`. Ela escolhe automaticamente o separador correto para o sistema em que o código está rodando, tornando seu programa compatível com qualquer plataforma.

Copie e Teste!

```python
import os

# Nome da pasta e do arquivo que queremos
pasta_dados = "dados"
arquivo_relatorio = "relatorio_final.csv"

# Construindo o caminho de forma segura
caminho_completo = os.path.join(pasta_dados, arquivo_relatorio)

print(f"O caminho construído é: {caminho_completo}")

# Você pode juntar vários componentes
caminho_mais_longo = os.path.join("ProjetoCITHA", "dados", "
    sensores", "log.txt")
print(f"Caminho mais complexo: {caminho_mais_longo}")
```

Saída Esperada

Saída Esperada (em Windows)

```
O caminho construído é: dados\relatorio_final.csv
Caminho mais complexo: ProjetoCITHA\dados\sensores\log.txt
```

Saída Esperada (em Linux/macOS)

```
O caminho construído é: dados/relatorio_final.csv
Caminho mais complexo: ProjetoCITHA/dados/sensores/log.txt
```

Com o entendimento sólido de arquivos e caminhos, agora estamos prontos para a parte prática: abrir esses arquivos para ler as informações que eles contêm e escrever nossos próprios dados para guardá-los de forma permanente.

2.2.2 Lendo e Escrevendo em Arquivos de Texto

Até agora, nossos programas funcionaram como uma conversa: as informações que criamos com variáveis e listas existem apenas enquanto o programa está em execução. Quando ele termina, tudo é esquecido. Mas e se quiséssemos criar um diário de campo para anotar as espécies de aves que observamos, ou registrar os dados de um sensor de qualidade da água de um igarapé ao longo de vários dias? Para isso, nosso programa precisa de uma ”memória”permanente. Ele precisa aprender a escrever em um caderno. No mundo digital, esse caderno é um **arquivo de texto**.

Nesta seção, vamos aprender a habilidade fundamental de salvar nossos dados em arquivos `.txt` e, depois, lê-los de volta. Esta é a forma mais direta de persistência de dados, transformando nossos programas de meros executores de tarefas em verdadeiros guardiões de informação.

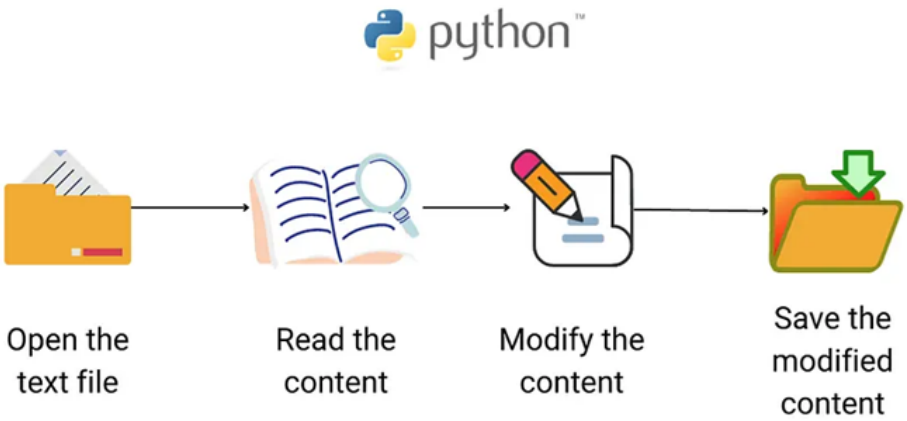

Figura 2.4: Fluxo de ações envolvendo arquivos de texto em Python
Fonte: Ravish Kumar, 2024

Abrindo e Escrevendo em um Arquivo: O Bloco `with open`

A maneira mais segura e recomendada de trabalhar com arquivos em Python é usando o bloco `with open()`. Pense nele como um assistente cuidadoso: ele abre o arquivo para você, permite que você faça o que precisa (ler ou escrever) e, o mais importante, garante que o arquivo seja fechado corretamente no final, mesmo que ocorra um erro.

Para escrever em um arquivo, usamos o modo de abertura `'w'` (do inglês, *write*). Este modo diz ao Python para criar um novo arquivo (ou apagar completamente um existente) para que possamos escrever nele.

Caso Prático

Vamos criar nosso primeiro registro permanente. Imagine que estamos instalando uma nova estação de monitoramento ambiental em uma reserva. Nossa primeira tarefa é criar um arquivo de log para registrar o início das operações.

Copie e Teste!

```
# O nome do arquivo que queremos criar
nome_arquivo = "log_estacao_1.txt"

# Usamos 'with open' para abrir o arquivo em modo de escrita ('w')
# A variável 'arquivo' nos dá acesso para interagir com o arquivo
with open(nome_arquivo, 'w') as arquivo:
    # O método .write() escreve uma string no arquivo
    arquivo.write("Registro de operacoes da Estacao de
   Monitoramento Alpha.\n")
    arquivo.write("Dados coletados a partir de 24/07/2025.")

print(f"Arquivo '{nome_arquivo}' criado com sucesso!")
```

Saída Esperada

```
Arquivo 'log_estacao_1.txt' criado com sucesso!
```

Após executar este código, um novo arquivo chamado `log_estacao_1.txt` aparecerá na mesma pasta do seu script Python. Se você abri-lo, verá o seguinte conteúdo:

Saída Esperada

```
Registro de operacoes da Estacao de Monitoramento Alpha.
Dados coletados a partir de 24/07/2025.
```

O método `.write()` por padrão não pula linha quando realizamos uma nova chamada da função `print()`, portanto, foi necessário adicionar a quebra de linha manualmente através do caractere `\n` ao final da primeira string.

Fique Alerta!

Cuidado: O modo `'w'` é destrutivo! Se o arquivo `log_estacao_1.txt` já existisse, seu conteúdo antigo seria **completamente apagado** e substituído pelo novo. Sempre tenha certeza de que deseja sobrescrever um arquivo antes de usar o modo `'w'`.

Adicionando Informações: O Modo de Anexação (`'a'`)

Sobrescrever um arquivo nem sempre é o que queremos. Em um diário de campo, por exemplo, não apagamos as anotações antigas para escrever as novas; nós as adicionamos no final. Para este comportamento, usamos o modo de anexação, ou *append*, representado pela letra `'a'`.

Caso Prático

Nossa equipe de biólogos fez novas observações de campo e precisa adicioná-las a um diário. O arquivo `diario_fauna.txt` já existe e contém registros anteriores. Precisamos adicionar as novas observações sem apagar as antigas.

Copie e Teste!

```
# Lista de novas observacoes
novas_observacoes = [
    "Observado um bando de araras-canga (Ara macao).",
    "Rastros de onca-pintada (Panthera onca) perto do igarape.",
    "Ninho de uirapuru-verdadeiro (Cyphorhinus arada) encontrado",
    "Fui picado..."
]

# Abrindo o arquivo em modo de anexacao ('a')
with open("diario_fauna.txt", 'a') as diario:
    # Adicionando um separador para organizar
    diario.write("\n--- Novas Observacoes ---\n")
    # Percorrendo a lista e escrevendo cada nova observacao
    for obs in novas_observacoes:
        diario.write(obs + "\n")

print("Diario de campo atualizado!")
```

Se o arquivo `diario_fauna.txt` continha ”Registro inicial.”antes da execução, após executar o script, seu conteúdo será:

Saída Esperada

```
Registro inicial.

--- Novas Observacoes ---
Observado um bando de araras-canga (Ara macao).
Rastros de onca-pintada (Panthera onca) perto do igarape.
Ninho de uirapuru-verdadeiro (Cyphorhinus arada) encontrado.
Fui picado...
```

O modo `'a'` é considerada uma escolha segura para adicionar informações a um arquivo de log ou a qualquer registro contínuo, já que os dados são incluídos de forma incremental, garantindo que os dados históricos sejam preservados (Matthes, 2023).

Lendo o Conteúdo de um Arquivo

Agora que sabemos como guardar informações, precisamos aprender a recuperá-las. Para isso, abrimos o arquivo em modo de leitura, ou *read*, representado pela letra `'r'`. Este é o modo padrão, mas é uma boa prática especificá-lo para maior clareza. Existem algumas maneiras de ler o conteúdo de um arquivo.

1. Lendo o Arquivo Inteiro: `.read()`

O método `.read()` lê todo o conteúdo do arquivo e o retorna como uma **única string**. Útil para arquivos pequenos, mas pode consumir muita memória se o arquivo for muito grande.

Copie e Teste!

```
# Lendo o conteudo completo do nosso diario
with open("diario_fauna.txt", 'r') as diario:
    conteudo_completo = diario.read()

print("--- Conteudo completo do Diario ---")
print(conteudo_completo)
```

Saída Esperada

```
--- Conteudo completo do Diario ---
Registro inicial.

--- Novas Observacoes ---
Observado um bando de araras-canga (Ara macao).
Rastros de onca-pintada (Panthera onca) perto do igarape.
Ninho de uirapuru-verdadeiro (Cyphorhinus arada) encontrado.
Fui picado...
```

2. Lendo Linha por Linha: A Abordagem Mais Eficiente

A forma mais comum e eficiente em Python para ler um arquivo é iterar sobre ele linha por linha com um laço for. Esta abordagem é ideal para arquivos grandes, pois lê apenas uma linha para a memória de cada vez. Mas o que isso realmente significa na prática?

Imagine que um sensor de monitoramento na Amazônia gera um arquivo de log com milhões de registros de temperatura, um por linha. Se tentássemos usar um método como .read() para carregar esse arquivo inteiro, poderíamos facilmente esgotar a memória do computador, fazendo o programa travar.

Ao usar um laço for, o Python adota uma estratégia de leitura preguiçosa (lazy reading). Ele não carrega o arquivo todo de uma vez. Em vez disso, a cada passagem do laço, ele vai até o disco, busca a próxima linha, a entrega para o seu código processar e, em seguida, a descarta da memória para buscar a próxima. Isso garante que seu programa possa processar arquivos de qualquer tamanho — de alguns kilobytes a muitos gigabytes — com a mesma eficiência e baixo uso de memória, tornando seu código mais robusto e escalável.

Caso Prático

Um sensor de temperatura nos enviou um arquivo `temperaturas.txt` com uma leitura por linha. Precisamos ler este arquivo e identificar qualquer temperatura acima de 35°C, que indica uma possível onda de calor na floresta.

Copie e Teste!

```
# Conteudo do arquivo temperaturas.txt (imagine que ele ja existe)
    :
# 32.5
# 34.1
# 36.2
# 33.8
# 35.5

print("Analisando temperaturas em busca de alertas:")
with open("temperaturas.txt", 'r') as f:
    for linha in f:
        # .strip() remove espacos em branco e novas linhas ('\n')
        temperatura_str = linha.strip()

        # Convertemos a string para um numero float
        temperatura_float = float(temperatura_str)

        if temperatura_float > 35.0:
            print(f"ALERTA: Temperatura elevada detectada! -> {
   temperatura_float} C")
```

Saída Esperada

```
Analisando temperaturas em busca de alertas:
ALERTA: Temperatura elevada detectada! -> 36.2 C
ALERTA: Temperatura elevada detectada! -> 35.5 C
```

Esta abordagem combina leitura de arquivo com a lógica de programação que já conhecemos (laços, condicionais, conversão de tipo), mostrando como a manipulação de arquivos se integra perfeitamente ao nosso fluxo de trabalho para resolver problemas práticos e significativos.

2.2.3 Trabalhando com Dados Estruturados: CSV

No tópico anterior, aprendemos a usar arquivos de texto como um caderno de anotações, perfeito para salvar um diário de campo ou registros sequenciais. Mas, e se nossos dados tiverem uma estrutura mais organizada, como uma tabela? Imagine um pesquisador na Amazônia que coleta dados sobre árvores: para cada uma, ele anota a espécie, a altura, o diâmetro do tronco e as coordenadas geográficas. Tentar salvar isso em um arquivo `.txt` simples seria confuso e difícil de processar depois.

É para isso que servem os arquivos **CSV (Comma-Separated Values, ou Valores Separados por Vírgula)**. Pense neles não como um caderno, mas como uma planilha digital. São arquivos de texto com uma regra simples: cada linha representa uma fileira da tabela e as vírgulas separam os valores de cada coluna. Este formato é universal, legível tanto por humanos quanto por máquinas, e é o padrão para exportar dados de programas como Excel e Google Sheets. Nesta seção, vamos aprender a dominar este formato com o módulo nativo do Python: `csv`.

Um arquivo CSV é a forma mais comum de armazenar dados tabulares. A sua estrutura é direta: a primeira linha geralmente contém os **cabeçalhos** (os nomes das colunas), e as linhas seguintes contêm os dados, onde cada valor é separado por um delimitador, que, por padrão, é uma vírgula.

	A	B	C
1	**local**	**ph**	**turbidez**
2	Encontro das Aguas	6.8	5.2
3	Comunidade Sao Jose	7.1	3.1
4	Frente a Manaus	6.5	8.9
5	Reserva Ducke	6.9	2.5

	A
1	**local,ph,turbidez**
2	Encontro das Aguas,6.8,5.2
3	Comunidade Sao Jose,7.1,3.1
4	Frente a Manaus,6.5,8.9
5	Reserva Ducke,6.9,2.5

Figura 2.5: Representação visual de dados tabulares (esquerda) e seu formato correspondente em arquivo CSV (direita).

Para interagir com esses arquivos de forma eficiente e segura, Python nos oferece o módulo `csv`, que ajuda a lida com diversas particularidades do formato de forma simples e otimizada, facilitando a vizualização e manipulação dos dados.

Lendo Arquivos CSV com `csv.reader`

A forma mais fundamental de ler um arquivo CSV é com o `csv.reader`. Ele processa o arquivo linha por linha e, para cada uma, nos devolve uma lista de strings, onde cada string é um valor daquela coluna.

Caso Prático

Uma equipe de monitoramento ambiental nos forneceu o arquivo para consulta nomeado como `qualidade_agua.csv`, contendo medições de pH e turbidez da água coletada em diferentes pontos do Rio Negro. Nossa primeira tarefa é ler e exibir esses dados.

Antes de começar a trabalhar com o arquivo `qualidade_agua.csv`, é considerada uma boa prática tentar verificar se o conteúdo do arquivo está integro:

Saída Esperada

```
local,ph,turbidez
Encontro das Aguas,6.8,5.2
Comunidade Sao Jose,7.1,3.1
Frente a Manaus,6.5,8.9
Reserva Ducke,6.9,2.5
```

Fique Alerta!

Quando trabalhamos com grandes volumes de dados, alguns editores de arquivos podem ter dificuldades de abrir arquivos muito grandes, consumindo muita memória RAM e causando lentidão e até travamentos. Por outro lado, IDE modernas podem ser menos simples e intuitivas do que ferramentas de edição de dados tradicionais, mas são mais robustas, resilientes e escaláveis para aplicação em projetos.

Agora que já verificamos o conteúdo do arquivo, vamos usar o `csv.reader` para manipular e processr seu conteúdo:

Copie e Teste!

```
import csv

nome_arquivo = 'qualidade_agua.csv'

print(f"Lendo dados do arquivo: {nome_arquivo}\n")
# O argumento newline='' é importante para evitar linhas em branco
    inesperadas
with open(nome_arquivo, mode='r', newline='', encoding='utf-8') as
    arquivo_csv:
    # Criamos um objeto leitor
    leitor_csv = csv.reader(arquivo_csv)

    # Iteramos sobre cada linha do leitor
    for linha in leitor_csv:
        print(linha)
```

Saída Esperada

```
Lendo dados do arquivo: qualidade_agua.csv

['local', 'ph', 'turbidez']
['Encontro das Aguas', '6.8', '5.2']
['Comunidade Sao Jose', '7.1', '3.1']
['Frente a Manaus', '6.5', '8.9']
['Reserva Ducke', '6.9', '2.5']
```

Note que o `csv.reader` nos devolveu cada linha como uma lista de strings. Embora funcional, essa abordagem tem uma fraqueza: para acessar um dado, precisamos usar seu índice (ex: `linha[1]` para o pH). Isso torna o código pouco legível e frágil. E se alguém alterar a ordem das colunas no arquivo? Nosso código quebraria.

A Melhor Abordagem: Lendo com `csv.DictReader`

Para resolver a fragilidade dos índices, o Python nos oferece uma ferramenta muito mais poderosa e "Pythônica": o `csv.DictReader`. Em vez de uma lista, ele lê cada linha e a transforma em um **dicionário**, onde as chaves são os nomes do cabeçalho.

Caso Prático

Utilizando o mesmo arquivo `qualidade_agua.csv`, nossa nova tarefa é analisar os dados e emitir um alerta para qualquer local cujo nível de turbidez da água seja superior a 5.0, o que pode indicar poluição ou excesso de sedimentos.

Copie e Teste!

```
import csv

nome_arquivo = 'qualidade_agua.csv'

print("Analisando niveis de turbidez...\n")
with open(nome_arquivo, mode='r', newline='', encoding='utf-8') as
    arquivo_csv:
    # Criamos um leitor que transforma cada linha em um dicionario
    leitor_dict = csv.DictReader(arquivo_csv)

    for linha in leitor_dict:
        # Acessamos os dados pela chave (nome da coluna)
        local = linha['local']
        turbidez = float(linha['turbidez']) # Nao esqueca de
   converter para numero!

        if turbidez > 5.0:
            print(f"ALERTA: Turbidez elevada em '{local}'. Valor:
   {turbidez}")
```

Saída Esperada

```
Analisando niveis de turbidez...

ALERTA: Turbidez elevada em 'Encontro das Aguas'. Valor: 5.2
ALERTA: Turbidez elevada em 'Frente a Manaus'. Valor: 8.9
```

O código agora é muito mais legível e robusto. Acessar `linha['turbidez']` é mais claro que `linha[2]`, e se a ordem das colunas no arquivo mudar, nosso código continuará funcionando perfeitamente. Como aponta Matthes (2023), usar as ferramentas certas para a estrutura de dados correta é uma marca de um programador eficaz.

Escrevendo em Arquivos CSV

Assim como é possível ler, também é possível escrever dados em arquivos CSV. Isso é extremamente útil para salvar os resultados de nossas análises ou para criar conjuntos de dados que podem ser facilmente abertos em outros programas. Para isso, utilizamos o método `csv.writer` para facilitar essa escrita e manipulação dos arquivos.

Caso Prático

Após uma expedição, nossa equipe catalogou várias espécies de peixes da bacia amazônica. Os dados estão em uma lista de listas dentro do nosso programa. Precisamos salvar esses dados em um arquivo chamado `peixes_catalogados.csv`.

Copie e Teste!

```
import csv

# Nossos dados: a primeira lista é o cabeçalho
dados_peixes = [
    ['nome_popular', 'nome_cientifico', 'risco_extincao'],
    ['Pirarucu', 'Arapaima gigas', 'Vulneravel'],
    ['Tambaqui', 'Colossoma macropomum', 'Quase Ameacado'],
    ['Tucunare', 'Cichla ocellaris', 'Pouco Preocupante']
]

nome_arquivo = 'peixes_catalogados.csv'

# Abrimos o arquivo em modo de escrita ('w')
with open(nome_arquivo, mode='w', newline='', encoding='utf-8') as
    arquivo_csv:
    escritor_csv = csv.writer(arquivo_csv)

    # .writerows() escreve todas as listas de uma vez
    escritor_csv.writerows(dados_peixes)

print(f"Arquivo '{nome_arquivo}' salvo com sucesso!")
```

Se você abrir o arquivo `peixes_catalogados.csv` vai encontrar um arquivo CSV perfeitamente formatado, pronto para ser utilizado em uma planilha ou outra análise.

Fique Alerta!

Assim como `open` com o modo `'w'` para arquivos de texto, usar `csv.writer` em um arquivo existente irá **sobrescrevê-lo completamente**. Se você precisar adicionar novas linhas a um CSV existente, abra o arquivo em modo de anexação (`'a'`).

Dominar a manipulação de arquivos CSV com o módulo `csv` abre um mundo de possibilidades para a análise de dados. É a ponte entre a lógica interna dos seus programas em Python e o vasto ecossistema de dados tabulares utilizados em ciência, negócios e tecnologia.

2.2.4 Trabalhando com Dados Estruturados: JSON

No tópico anterior, exploramos os arquivos CSV, que são perfeitos para organizar dados em formato de tabela, como uma planilha. Mas e se nossos dados não se encaixarem bem em linhas e colunas? E se precisarmos de uma estrutura mais flexível, com informações aninhadas, como uma ficha de catalogação de uma espécie que contém uma lista de locais onde foi encontrada e um dicionário de suas características?

É para resolver esse desafio que utilizamos o formato **JSON (JavaScript Object Notation)**. Se um CSV é como uma planilha, o JSON é como um documento de texto super organizado ou um cartão de contato digital, onde cada informação tem um rótulo claro e podemos guardar listas dentro de listas ou informações dentro de outras. Embora o nome venha da linguagem JavaScript, o JSON é um formato de texto completamente independente de linguagem e se tornou o padrão universal para a troca de dados na internet, sendo a base para o funcionamento de praticamente todos os aplicativos e sites modernos que você usa.

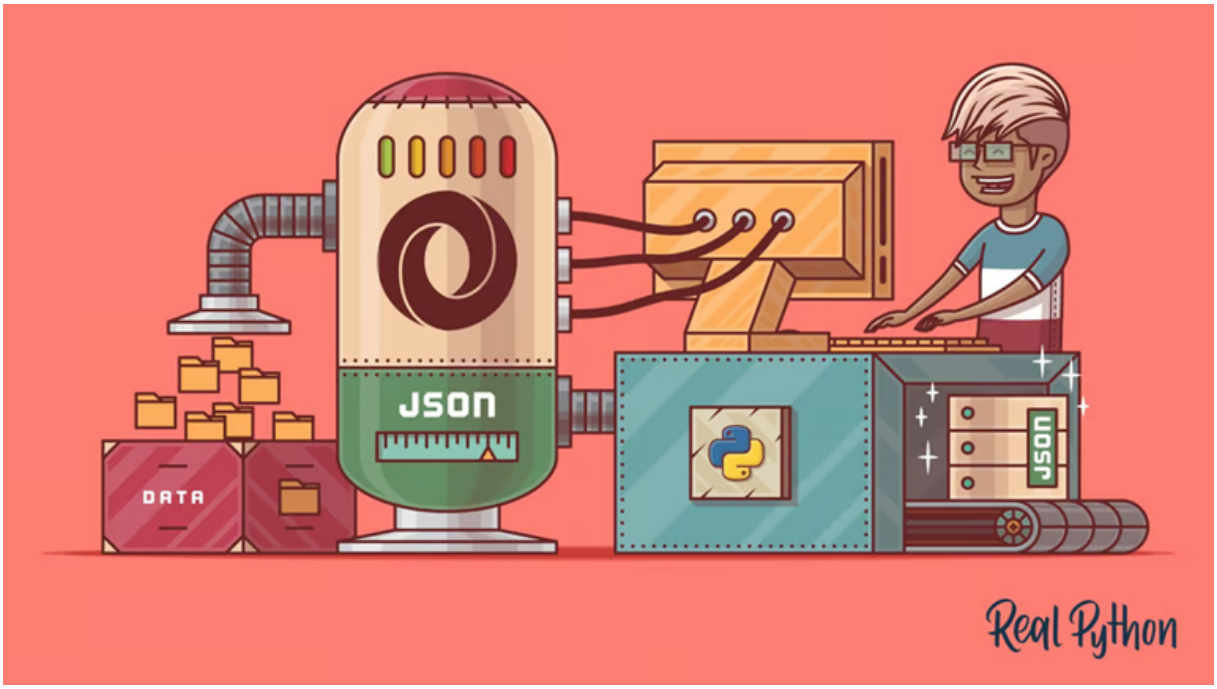

Figura 2.6: Arte que ilustra a conversão entre JSON e Python
Fonte: Mehedi Hasan, 2024

O que é JSON e por que é tão importante?

JSON é uma forma leve e legível de representar dados estruturados. Sua grande vantagem é que sua estrutura se alinha quase perfeitamente com duas das estruturas de dados mais importantes do Python: os **dicionários** e as **listas**. As regras do JSON são simples:

- Dados são organizados em pares de **chave-valor**, assim como nos dicionários do Python. As chaves são sempre strings entre aspas duplas.
- Coleções de pares chave-valor formam um **objeto** (equivalente a um dicionário Python), delimitado por chaves `{}`.
- Listas ordenadas de valores formam um **array** (equivalente a uma lista Python), delimitado por colchetes `[]`.
- Os valores podem ser strings, números, booleanos (`true`/`false`), arrays ou até mesmo outros objetos, permitindo a criação de estruturas de dados complexas e aninhadas.

Essa flexibilidade torna o JSON a linguagem preferida para as **APIs (Application Programming Interfaces)**, que são as pontes de comunicação que permitem que diferentes sistemas troquem informações. Quando seu aplicativo de clima mostra a previsão para Manaus, ele provavelmente recebeu esses dados de um servidor em formato JSON.

Lendo Arquivos JSON com o Módulo `json`

O Python possui uma biblioteca nativa fantástica chamada `json`, que torna a conversão entre o formato JSON e os objetos Python incrivelmente simples. Para ler um arquivo JSON e transformá-lo em um dicionário ou lista, usamos a função `json.load()`.

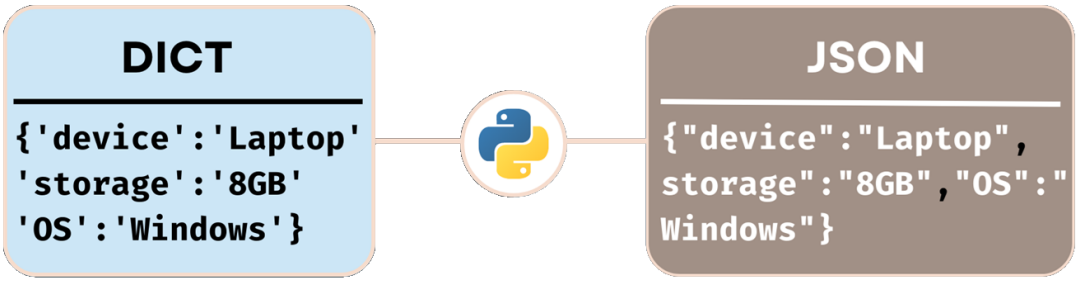

Figura 2.7: Conversão de dicionário para JSON em Python
Fonte: Adaptado de Ayushi Trivedi, 2024

O processo de leitura de um arquivo JSON em Python é notavelmente direto e segue uma lógica de duas etapas. Primeiro, abrimos o arquivo da mesma forma que faríamos com um arquivo de texto comum, usando o bloco `with open()`, que garante que o arquivo seja manuseado de forma segura e fechado automaticamente. Em seguida, passamos o objeto do arquivo para a função `json.load()`, que atua como um tradutor inteligente. Ela lê o texto formatado em JSON, reconhece sua estrutura de objetos, arrays e pares de chave-valor, e converte tudo isso em um objeto Python nativo — geralmente um dicionário ou uma lista — que podemos manipular facilmente no restante do nosso código.

Caso Prático

Uma estação de monitoramento autônoma na Reserva Florestal Ducke grava dados em um arquivo, `sensor_igarap_v1.json`, com a última leitura da qualidade da água de um igarapé. Precisamos de um programa para ler esse arquivo, processar os dados e exibir um resumo das condições atuais.

Vamos verificar o conteúdo do nosso arquivo `sensor_igarap_v1.json`, para ter noção do que devemos encontrar depois da importação:

Saída Esperada

```
{
  "id_sensor": "IGRP-001",
  "local": "Reserva Florestal Ducke",
  "timestamp": "2025-07-25T09:30:00Z",
  "ativo": true,
  "leituras": [
    {"parametro": "pH", "valor": 6.8, "unidade": ""},
    {"parametro": "Turbidez", "valor": 4.5, "unidade": "NTU"},
    {"parametro": "Oxigenio Dissolvido", "valor": 7.2, "unidade":
       "mg/L"}
  ]
}
```

Agora que já verificamos o conteúdo do arquivo, vamos usar Python para ler e interpretar esses dados.

Copie e Teste!

```
import json

nome_arquivo = 'sensor_igarap_v1.json'

print(f"--- Lendo dados do sensor do arquivo: {nome_arquivo} ---")

with open(nome_arquivo, mode='r', encoding='utf-8') as
    arquivo_json:
  # json.load() lê o arquivo e converte o JSON para um objeto
    Python
  dados_sensor = json.load(arquivo_json)

# Agora 'dados_sensor' é um dicionário Python comum!
local = dados_sensor['local']
id_sensor = dados_sensor['id_sensor']
print(f"Local do Sensor: {local} (ID: {id_sensor})")

print("\nÚltimas Leituras Registradas:")
# Podemos iterar sobre a lista de leituras
for leitura in dados_sensor['leituras']:
  param = leitura['parametro']
  val = leitura['valor']
  unid = leitura['unidade']
  print(f" -> {param}: {val} {unid}")
```

Saída Esperada

```
--- Lendo dados do sensor do arquivo: sensor_igarap_v1.json
    ---
Local do Sensor: Reserva Florestal Ducke (ID: IGRP-001)

Últimas Leituras Registradas:
 -> pH: 6.8
 -> Turbidez: 4.5 NTU
 -> Oxigenio Dissolvido: 7.2 mg/L
```

Escrevendo Dados em Arquivos JSON

O processo inverso também é fundamental. Muitas vezes, nosso programa irá coletar ou processar dados que precisam ser salvos em um formato estruturado para serem usados mais tarde por outro sistema. Para isso, convertemos um dicionário ou lista Python em um arquivo JSON usando a função `json.dump()`.

Caso Prático

Nossa equipe de campo catalogou uma nova espécie de planta, a "Vitória-Régia"(*Victoria amazonica*). Precisamos criar um dicionário Python com as informações coletadas e salvá-lo em um novo arquivo, `ficha_especie_001.json`, para adicionar ao banco de dados do projeto.

Copie e Teste!

```
import json

nova_ficha = {
  "id_especie": "FLR-001",
  "nome_popular": "Vitória-Régia",
  "nome_cientifico": "Victoria amazonica",
  "familia": "Nymphaeaceae",
  "coletor": "A. Silva",
  "locais_avistados": [
    {"local": "Lago Janauari", "coords": [-3.456, -60.123]},
    {"local": "Rio Solimões", "coords": [-3.123, -59.456]}
  ]
}

nome_arquivo = 'ficha_especie_001.json'

with open(nome_arquivo, mode='w', encoding='utf-8') as
    arquivo_json:
  # json.dump() escreve o objeto Python no arquivo em formato JSON
  # 'indent=4' formata o arquivo de forma legível para humanos
  json.dump(nova_ficha, arquivo_json, indent=4, ensure_ascii=False
    )

print(f"Ficha da espécie salva com sucesso no arquivo '{
    nome_arquivo}'!")
```

Fique Alerta!

Note os parâmetros extras em `json.dump()`:

- `indent=4`: Este argumento é seu melhor amigo ao criar arquivos JSON. Ele instrui a função a formatar o arquivo com uma indentação de 4 espaços, tornando-o imensamente mais fácil de ler para um ser humano. Sem ele, todo o conteúdo seria salvo em uma única linha.
- `ensure_ascii=False`: Por padrão, o módulo `json` converte caracteres não-ASCII (como "é"e "ç") em sequências de escape (ex: `\u00e9`). Utilizar a configuração `ensure_ascii=False` garante que os caracteres especiais sejam salvos corretamente, o que é essencial ao trabalhar com textos referentes a língua portuguesa.

Após executar o código, um novo arquivo chamado `ficha_especie_001.json` será criado. Se você abri-lo, verá o conteúdo perfeitamente formatado:

Saída Esperada

Conteúdo do arquivo `ficha_especie_001.json`

```
{
    "id_especie": "FLR-001",
    "nome_popular": "Vitória-Régia",
    "nome_cientifico": "Victoria amazonica",
    "familia": "Nymphaeaceae",
    "coletor": "A. Silva",
    "locais_avistados": [
        {
            "local": "Lago Janauari",
            "coords": [
                -3.456,
                -60.123
            ]
        },
        {
            "local": "Rio Solimões",
            "coords": [
                -3.123,
                -59.456
            ]
        }
    ]
}
```

Dominar a manipulação de JSON com Python, conforme demonstrado em obras práticas como a de Matthes (2023), é uma habilidade essencial. Ela não só permite que seus programas salvem e carreguem dados complexos de forma robusta, mas também abre as portas para a interação com o vasto mundo das APIs e serviços web, tornando suas aplicações muito mais poderosas e conectadas.

2.3 Aplicando seus Conhecimentos

Até agora, você aprendeu a organizar seu código em blocos reutilizáveis com **funções** e a dar ”memória”aos seus programas, ensinando-os a ler e escrever em **arquivos**. Essas são duas das habilidades mais poderosas na programação, pois permitem criar aplicações complexas, organizadas e que interagem com o mundo real.

Agora é a hora de consolidar esse conhecimento. Os exercícios a seguir foram projetados para desafiá-lo a combinar essas habilidades para resolver problemas práticos. Você criará funções para processar dados, salvar seus resultados em diferentes formatos e carregar informações de arquivos para dentro de seus programas. Lembre-se: programar é como aprender um idioma; a fluência vem com a prática.

2.3.1 Exercícios com Funções

1. **Calculadora de Densidade Demográfica:** A densidade demográfica é um indicador importante para estudos geográficos e ambientais na Amazônia. Crie uma função chamada `calcular_densidade` que receba dois parâmetros: a `populacao` (um número inteiro) e a `area_km2` (um número float). A função deve calcular e retornar a densidade demográfica (população / área).

   - **Dica:** A fórmula é `densidade = populacao / area_km2`.
   - **Teste sua função:** Chame a função com os dados de uma cidade como Manaus (população aproximada de 2.200.000 e área de 11.401 km²) e imprima o resultado formatado.

2. **Classificador de pH do Solo:** O pH é uma medida crucial para a agricultura. Crie uma função `classificar_ph` que recebe um valor de pH (um float) e retorna uma string com a classificação do solo, de acordo com as seguintes regras:

   - Se pH < 5.5: ”Solo Muito Ácido”
   - Se 5.5 <= pH < 6.5: ”Solo Ácido”
   - Se 6.5 <= pH <= 7.5: ”Solo Neutro”
   - Se pH > 7.5: ”Solo Alcalino”

   Teste sua função: Chame a função com diferentes valores de pH (ex: 4.8, 6.0, 7.2, 8.1) e imprima os resultados.

3. **Análise de Dados de um Sensor:** Você recebeu uma lista com as medições de temperatura de um sensor ao longo de um dia: `temperaturas = [28.5, 29.1, 31.2, 33.0, 34.9, 32.1, 30.8, 29.5]`. Crie uma função `analisar_leituras` que receba uma lista de números e retorne a temperatura mínima, a máxima e a média da lista.

   - **Dica:** Python já possui as funções `min()`, `max()` e `sum()`. Para a média, lembre-se de que é a soma dos valores dividida pela quantidade de valores (`len()`).
   - **Retorno Múltiplo:** Sua função deve retornar os três valores de uma só vez.

4. **Ordenando Dados de Coleta com Lambda:** Uma equipe de biólogos coletou dados sobre diferentes espécies e os armazenou em uma lista de dicionários. Eles precisam ordenar a lista com base na quantidade de indivíduos avistados, do maior para o menor.

   Copie e Teste!

   ```
   coletas = [
       {'especie': 'Capivara', 'quantidade': 12},
       {'especie': 'Tucano', 'quantidade': 25},
       {'especie': 'Onça-pintada', 'quantidade': 2},
       {'especie': 'Arara', 'quantidade': 18}
   ]
   ```

   Desafio: Use o método `.sort()` (ou a função `sorted()`) com uma função `lambda` como chave (`key`) para ordenar a lista `coletas`.

2.3.2 Exercícios com Arquivos de Texto (.txt)

5. **Diário de Bordo Digital:** Crie um programa que funcione como um ”Diário de Bordo”. O programa deve:

   (a) Perguntar ao usuário o nome do arquivo de log que ele deseja criar

   (b) Entrar em um laço onde o usuário pode digitar várias entradas para o diário.

   (c) Cada entrada digitada pelo usuário deve ser salva em uma nova linha no arquivo.

   (d) O laço termina quando o usuário digitar a palavra ”SAIR”.

   Dica: Use o modo de anexação (`'a'`) para não apagar as entradas anteriores a cada nova execução. Adicione a data e a hora em cada registro para um diário mais completo!

6. **Contador de Alertas em Log:** Um sensor de monitoramento ambiental gera um arquivo de log (`sensor_log.txt`). Precisamos de um programa que leia este arquivo e conte quantas vezes a palavra ”ALERTA”aparece.

   Conteúdo de `sensor_log.txt` (crie este arquivo para testar):

   Saída Esperada

   ```
   2025-07-25 10:00 - Status: OK
   2025-07-25 10:05 - Status: OK
   2025-07-25 10:10 - ALERTA: Nível de água subindo rapido.
   2025-07-25 10:15 - Status: OK
   2025-07-25 10:20 - ALERTA: Temperatura acima do limite.
   ```

   Seu programa deve ler o arquivo linha por linha e, ao final, imprimir: ”Total de alertas encontrados: 2”.

2.3.3 Exercícios com Arquivos CSV

7. **Exportando Dados de Alunos para CSV:** Crie um programa que salve os dados de alunos em uma lista de dicionários em um arquivo chamado `alunos.csv`.

   Copie e Teste!

   ```
   dados_alunos = [
       {'id': 'A001', 'nome': 'Ana Silva', 'curso': 'Informática
   '},
       {'id': 'A002', 'nome': 'Bruno Costa', 'curso': '
   Agropecuária'},
       {'id': 'A003', 'nome': 'Carla Dias', 'curso': 'Meio
   Ambiente'}
   ]
   ```

   Dica: Use o módulo `csv` e o `csv.DictWriter` para facilitar a escrita, pois ele pode usar as chaves do dicionário para criar o cabeçalho automaticamente.

8. **Análise de Produção de Açaí:** Você recebeu um arquivo `producao_acai.csv` com dados da produção de açaí (em toneladas) de uma cooperativa. Sua tarefa é ler o arquivo e calcular a produção total e a média mensal.

   Conteúdo de `producao_acai.csv` (crie este arquivo para testar):

   Saída Esperada

   ```
   mes,producao_toneladas
   Janeiro,15.5
   Fevereiro,12.8
   Março,18.2
   Abril,17.9
   ```

   Seu programa deve ler o arquivo (usando `csv.DictReader`), somar todos os valores da coluna `producao_toneladas` e calcular a média. Ao final, imprima os resultados.

2.3.4 Exercícios com Arquivos JSON

9. **Salvando Configurações de um Projeto:** As configurações de um sistema de monitoramento são armazenadas em um dicionário Python. Crie um programa que salve esse dicionário em um arquivo `config.json` de forma bem formatada.

   Copie e Teste!

   ```
   configuracoes = {
       "id_projeto": "CITHA-MON-01",
       "local": "Reserva de Desenvolvimento Sustentável Mamirauá
      ",
       "ativo": True,
       "sensores_instalados": ["Temperatura", "Umidade", "
      Qualidade do Ar"],
       "limites_alerta": {
           "temperatura_max": 38.0,
           "umidade_min": 60.0
       }
   }
   ```

   Dica: Use `json.dump()` com os argumentos `indent=4` para a formatação adequada e `ensure_ascii=False` para garantir que caracteres especiais que não sejam ASCII (”á”por exemplo) sejam salvos corretamente.

10. **Lendo Ficha de Espécie em JSON:** Um biólogo enviou a ficha de uma espécie em formato JSON. Crie um programa que leia o arquivo `especie.json` e imprima uma frase resumindo as informações.

    Conteúdo de `especie.json` (crie este arquivo para testar):

Saída Esperada

```
{
  "nome_popular": "Boto-cor-de-rosa",
  "nome_cientifico": "Inia geoffrensis",
  "habitat": "Rios de água doce da bacia amazônica",
  "status_conservacao": "Em perigo"
}
```

Saída esperada: "O Boto-cor-de-rosa (Inia geoffrensis), que vive em Rios de água doce da bacia amazônica, está com o status de conservação: Em perigo."

2.4 Considerações do Módulo 2

Conhecendo um pouco mais!

Materiais Práticos do Módulo 2
Lembre-se que todos os scripts, datasets e o notebook para este módulo estão disponíveis em nosso repositório no GitHub. Para Instruções detalhadas sobre como utilizar os materiais, consulte a subseção 1.5 Considerações do Módulo 1

- **Scripts e Datasets do Módulo 2:**
  `https://github.com/CITHA-AM/Python/tree/main/Modulo%202`
- **Notebook do Módulo 2 (Colab):**
  `https://colab.research.google.com/github/CITHA-AM/Python/blob/main/Modulo%202.ipynb`

Parabéns! Ao concluir o Módulo 2, você deu um dos passos mais significativos na sua jornada como desenvolvedor: a transição de escrever instruções sequenciais para construir aplicações verdadeiramente estruturadas e com memória.

Neste capítulo, mergulhamos em dois pilares que sustentam a programação robusta. O primeiro foi a modularização, onde as funções se revelaram nossas principais ferramentas. Aprendemos a encapsular a complexidade em blocos de código nomeados, reutilizáveis e fáceis de manter, seguindo o princípio fundamental de Don't Repeat Yourself (DRY). Você agora entende como gerenciar a comunicação entre diferentes partes do seu código por meio de parâmetros e valores de retorno, como as variáveis se comportam em escopos locais e globais e a importância de criar funções claras e bem documentadas com docstrings e type hints.

O segundo pilar foi a persistência de dados. Vimos como quebrar as barreiras da memória volátil, ensinando nossos programas a ler e a escrever em arquivos. Essa habilidade permite que suas aplicações salvem resultados, carreguem configurações e processem informações de fontes externas, tornando-as infinitamente mais úteis e dinâmicas. Você explorou desde o manuseio de simples arquivos de texto (.txt), ideais para logs e registros, até formatos estruturados como CSV e JSON, que são a espinha dorsal da troca de dados no mundo da tecnologia atual.

Ao dominar a organização do código com funções e a manipulação de arquivos, você não está mais apenas criando scripts que executam e são esquecidos. Você agora é capaz

de desenvolver programas que podem crescer em complexidade de forma organizada e que ”lembram”informações entre execuções.

Com essa base sólida, você está perfeitamente preparado para o próximo desafio. No Módulo 3, utilizaremos essas habilidades para explorar um dos campos mais fascinantes da programação: a Análise de Dados. Vamos aprender a usar bibliotecas poderosas como Pandas e Matplotlib para extrair, manipular e visualizar dados de forma eficiente, transformando informações brutas em insights valiosos. Continue com a dedicação, pois o que você aprendeu aqui será a base para todas as análises que faremos a seguir.

Capítulo 3

Análise de Dados com Pandas e Matplotlib

Iniciando o diálogo...

Parabéns por chegar até aqui! Nos módulos anteriores, você aprendeu a construir programas estruturados com funções e a dar ”memória”a eles, salvando e lendo informações em arquivos. Você construiu a base e as paredes da sua casa. Agora, vamos aprender a entender o que acontece dentro dela, transformando os dados brutos que coletamos em conhecimento valioso. Este capítulo é a sua porta de entrada para o universo da Análise de Dados, uma das áreas mais impactantes da tecnologia. Usaremos duas das ferramentas mais poderosas do ecossistema Python, as bibliotecas **Pandas** e **Matplotlib**, para aprender a questionar, limpar, analisar e, o mais importante, contar as histórias que os dados escondem.

Imagine que você é um(a) biólogo(a) retornando de uma expedição de campo na Amazônia com dezenas de cadernos de anotações. Esses cadernos estão repletos de dados brutos: medições da qualidade da água, contagem de espécies, coordenadas geográficas, datas e horas. Essa pilha de anotações — nossos arquivos de dados — é valiosíssima, mas, em seu estado atual, é caótica e difícil de interpretar. Apenas ter os dados não é o suficiente; o verdadeiro poder está em extrair *insights* deles.

É exatamente aqui que a nossa jornada neste capítulo começa. Se os dados brutos são uma floresta densa e inexplorada, a biblioteca **Pandas** é o seu kit de exploração completo. Ela é a ferramenta que nos permite organizar esse caos. Com o Pandas, podemos pegar todas aquelas anotações espalhadas e organizá-las em uma tabela limpa e estruturada, chamada `DataFrame`. Pense no `DataFrame` como um mapa interativo da sua pesquisa. Com ele, você pode:

- **Inspecionar seus dados:** Ter uma visão geral do terreno que você está explorando.
- **Filtrar e selecionar:** Focar em uma área específica do mapa, como analisar apenas os dados de uma espécie ou de um local.
- **Limpar e corrigir:** Lidar com anotações borradas ou informações faltantes, garantindo a precisão do seu mapa.
- **Analisar e agregar:** Calcular estatísticas e agrupar informações para descobrir padrões ocultos, como o mês de maior incidência de uma certa espécie de ave.

Dominar o Pandas é o que transforma um amontoado de dados em uma fonte organizada e confiável de informações. É uma habilidade tão fundamental que seu criador, Wes McKinney, escreveu uma das obras de referência na área, consolidando o Pandas como a ferramenta padrão para manipulação de dados em Python (McKinney, 2018).

Contudo, um mapa cheio de números e coordenadas ainda não conta uma história de forma eficaz. Para compartilhar suas descobertas com o mundo, você precisa de algo mais visual. É aqui que entra a nossa segunda ferramenta poderosa: **Matplotlib**.

Se o Pandas é o seu kit de exploração e cartografia, **Matplotlib** é o seu estúdio de arte e comunicação. Ele permite que você transforme suas tabelas de dados e análises em gráficos e visualizações claras e impactantes. Com Matplotlib, você pode:

- Criar um **gráfico de linhas** para mostrar como a temperatura de um rio variou ao longo do ano.
- Gerar um **gráfico de barras** para comparar a população de diferentes espécies em uma reserva.
- Customizar cada detalhe, desde cores e legendas até títulos, para contar a história mais precisa e convincente possível.

A visualização de dados não é apenas sobre criar imagens bonitas; é uma ferramenta essencial de análise que nos permite identificar tendências, anomalias e padrões que seriam quase impossíveis de perceber apenas olhando para números. É a ponte entre a sua análise técnica e a compreensão humana.

Neste capítulo, você aprenderá a usar essas duas bibliotecas em conjunto. Primeiro, usaremos o Pandas para carregar, limpar e analisar conjuntos de dados do mundo real. Em seguida, usaremos o Matplotlib para dar vida a essas análises, criando visualizações que comunicam nossas descobertas de forma clara e profissional. Ao final, você estará equipado(a) não apenas para programar, mas para extrair significado de dados, uma habilidade essencial para enfrentar os desafios do século XXI.

3.1 Introdução à Biblioteca Pandas

No capítulo anterior, aprendemos a interagir com arquivos CSV e JSON, o que nos permitiu salvar e carregar dados de forma estruturada. No entanto, usar apenas as listas e dicionários do Python para analisar esses dados pode ser como tentar explorar a Amazônia com um mapa de papel e uma bússola: é possível, mas é trabalhoso, lento e fácil de se perder. Para navegar por grandes volumes de informação de forma eficiente, precisamos de uma ferramenta mais poderosa e especializada.

É aqui que entra a biblioteca **Pandas**. Pense nela como um sistema de GPS de última geração para seus dados. Ela foi criada especificamente para tornar a manipulação e a análise de dados não apenas mais fáceis, mas também incrivelmente rápidas. Dominar o Pandas é o que transforma um amontoado de dados em uma fonte organizada e confiável de informações. É uma habilidade tão fundamental que seu criador, Wes McKinney, escreveu uma das obras de referência na área, consolidando o Pandas como a ferramenta padrão para manipulação de dados em Python.

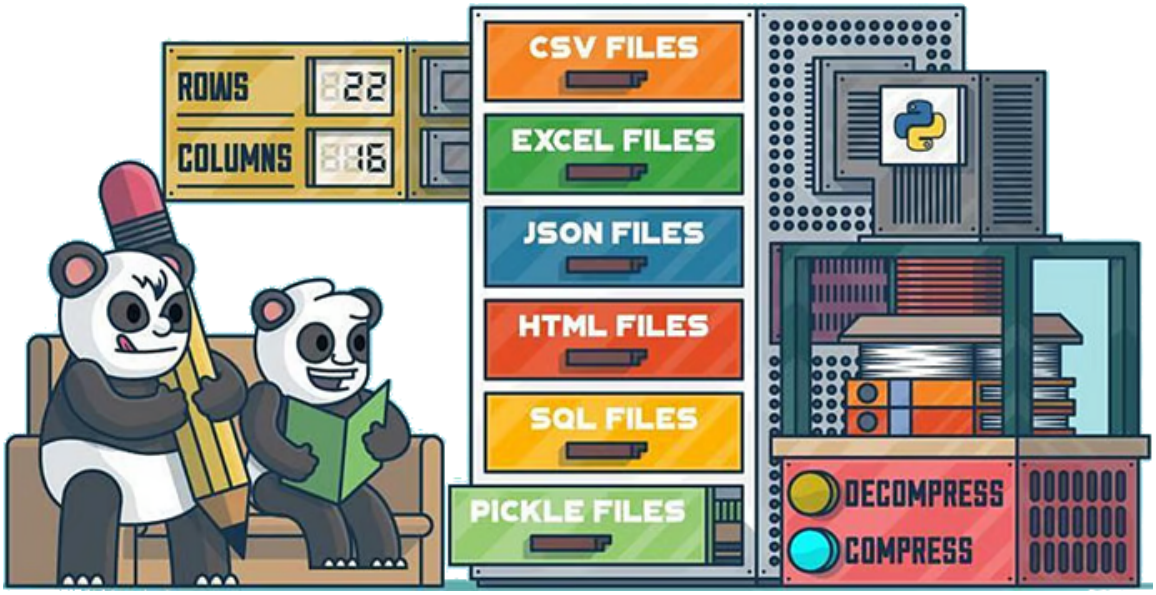

Figura 3.1: Arte que ilustra a abrangência da biblioteca Pandas
Fonte: Adaptado de Guilherme.py, 2021

Nesta seção, vamos dar os primeiros passos com essa ferramenta essencial. Começaremos entendendo por que bibliotecas como o Pandas são tão importantes e como elas aceleram nosso trabalho. Em seguida, conheceremos suas duas estruturas de dados fundamentais, o **DataFrame** (nossa tabela de dados superpoderosa) e a **Series** (a coluna que forma a tabela). Por fim, veremos como é surpreendentemente simples usar o Pandas para carregar os dados dos arquivos CSV que aprendemos a criar, trazendo-os para dentro do nosso ambiente de análise e preparando o terreno para a exploração.

3.1.1 O Poder das Bibliotecas: Por que usar Pandas?

No início da sua jornada com Python, você aprendeu a usar as ferramentas que vêm na ”caixa de ferramentas”padrão da linguagem: listas, dicionários, laços `for` e funções. Essas ferramentas são fantásticas e versáteis, mas, para certas tarefas especializadas, elas podem não ser as mais eficientes.

Imagine que você é um pesquisador monitorando a qualidade da água do Rio Negro e coletou dados em um arquivo CSV como este:

dados_rio.csv

```
local,ph,turbidez,temperatura_c
Encontro das Águas,6.8,5.2,28.5
Comunidade São José,7.1,3.1,29.1
Frente a Manaus,6.5,8.9,30.2
Reserva Ducke,6.9,2.5,27.8
```

Sua primeira tarefa é simples: calcular a temperatura média da água. Como faríamos isso usando apenas o que aprendemos até agora?

Caso Prático

Vamos tentar calcular a temperatura média da água usando as ferramentas padrão do Python, para entender as dificuldades que encontramos.

Copie e Teste!

```python
import csv

# Lista para guardar apenas os valores de temperatura
temperaturas = []

with open('dados_rio.csv', mode='r', encoding='utf-8') as arquivo:
    leitor_csv = csv.reader(arquivo)

    # Pula a primeira linha (o cabeçalho)
    next(leitor_csv)

    # Itera sobre as linhas para extrair a temperatura
    for linha in leitor_csv:
        # A temperatura está na 4ª coluna (índice 3)
        temperatura_str = linha[3]
        temperaturas.append(float(temperatura_str))

# Agora, calculamos a média manualmente
media = sum(temperaturas) / len(temperaturas)

print(f"A temperatura média da água é: {media:.2f}°C")
```

Saída Esperada

```
A temperatura média da água é: 28.90°C
```

Este código funciona, mas observe os desafios:

- **Muita preparação manual:** Tivemos que abrir o arquivo, criar um leitor, pular o cabeçalho manualmente, iterar linha por linha, extrair o valor de uma coluna específica usando seu índice (`linha[3]`), converter para `float` e só então calcular a média.
- **Código frágil:** E se alguém adicionasse uma nova coluna antes da temperatura? O índice `3` estaria errado e nosso código quebraria ou, pior, calcularia a média de uma coluna errada sem nos avisar.
- **Ineficiência:** Para tarefas mais complexas, como encontrar o local com a maior turbidez ou agrupar dados por região, nosso código se tornaria rapidamente longo, complicado e lento, especialmente com milhares ou milhões de linhas.

É aqui que o conceito de **bibliotecas** entra em jogo. Uma biblioteca em programação é um conjunto de códigos e funções pré-escritos que resolvem problemas comuns. Em vez de construir tudo do zero, você ”importa”uma biblioteca especializada e usa suas ferramentas otimizadas.

O **Pandas** é a biblioteca definitiva para análise de dados em Python. Ele foi projetado para lidar com dados tabulares (como os do nosso CSV) de forma intuitiva e extremamente eficiente. A principal estrutura de dados que o Pandas nos oferece é o **DataFrame**.

Pense em um DataFrame como uma versão superpoderosa de uma planilha do Excel ou do Google Sheets, diretamente no seu código Python. É uma tabela com linhas e colunas, mas com habilidades extraordinárias:

- **Colunas nomeadas:** Você acessa os dados pelo nome da coluna ao invés de um índice numérico frágil (ex: `dados['temperatura_c']`).
- **Tipos de dados inteligentes:** O Pandas detecta automaticamente se uma coluna contém números, textos ou datas, otimizando o armazenamento e os cálculos.
- **Funções prontas:** Quer a média de uma coluna? Basta usar `.mean()`. A mediana? `.median()`. O valor máximo? `.max()`. Tudo de forma direta e intuitiva.

Conhecendo um pouco mais!

O nome "Pandas"não vem do animal, mas sim do termo econométrico *"panel data"* (dados em painel), que se refere a conjuntos de dados que combinam observações ao longo do tempo para diferentes indivíduos. Isso reflete a origem da biblioteca, criada por Wes McKinney para facilitar a análise de dados financeiros.

Agora, vamos refazer nossa tarefa de calcular a temperatura média, mas desta vez usando o poder do Pandas.

Copie e Teste!

```
import pandas as pd

# 1. Ler o arquivo CSV diretamente para um DataFrame
# O 'pd' é o apelido padrão para o pandas
df = pd.read_csv('dados_rio.csv')

# 2. Calcular a média da coluna 'temperatura_c'
media_pandas = df['temperatura_c'].mean()

print(f"A temperatura média (calculada com Pandas) é: {
    media_pandas:.2f}°C")
```

Saída Esperada

```
A temperatura média (calculada com Pandas) é: 28.90°C
```

Veja a diferença! Em apenas duas linhas de código, o Pandas:

1. Abriu o arquivo.
2. Identificou o cabeçalho e nomeou as colunas.
3. Carregou todos os dados em uma estrutura organizada (o DataFrame `df`).
4. Calculou a média da coluna que especificamos pelo nome.

O código não é apenas mais curto, é imensamente mais legível, robusto e eficiente. É por isso que usamos bibliotecas: para nos apoiarmos no trabalho de especialistas e focarmos no que realmente importa, que é resolver nosso problema e extrair insights dos dados.

Fique Alerta!

Para usar uma biblioteca como o Pandas, primeiro você precisa instalá-la no seu ambiente Python. Se você estiver usando uma distribuição como o Anaconda, ele já vem incluído. Caso contrário, você pode instalá-lo facilmente abrindo seu terminal ou prompt de comando e digitando:

```
pip install pandas
```

Com o Pandas, a análise de dados deixa de ser uma tarefa de programação complexa e se torna uma exploração interativa. Nas próximas seções, vamos mergulhar nas estruturas de `DataFrame` e `Series` e aprender a usar as ferramentas que o Pandas oferece para filtrar, limpar e entender as histórias que nossos dados amazônicos têm a nos contar.

3.1.2 Series e DataFrames: As Estruturas Fundamentais

No tópico anterior, comparamos a biblioteca Pandas a um GPS de última geração para nossos dados. Agora, vamos conhecer os dois componentes que formam o coração desse sistema: a **Series** e o **DataFrame**. Pense neles como os blocos de construção fundamentais de qualquer análise. Se você estivesse construindo uma casa de LEGOs, a `Series` seria um único tipo de bloco, como um tijolo 2x1, e o `DataFrame` seria a estrutura completa que você monta com esses blocos, como uma parede ou a casa inteira.

Series 1
(especies_peixes)

Índice	Valores
Lago A	120
Lago B	85
Lago C	98

+

Series 2
(ph_agua)

Índice	Valores
Lago A	6.5
Lago B	6.8
Lago C	6.2

+

Series 3
(turbidez_ntu)

Índice	Valores
Lago A	4.5
Lago B	3.1
Lago C	5.8

Pandas DataFrame

Índice	especies_peixes	ph_agua	turbidez_ntu
Lago A	120	6.5	4.5
Lago B	85	6.8	3.1
Lago C	98	6.2	5.8

Figura 3.2: Diagrama da formação de um DataFrame a partir de Series (usando `tabular`).

Entender como essas duas estruturas funcionam e se relacionam é o passo mais importante para se tornar fluente em Pandas. Elas são a base sobre a qual todas as operações de limpeza, manipulação e análise de dados são construídas.

A `Series`: A Coluna Inteligente

A estrutura mais básica do Pandas é a `Series`. A maneira mais fácil de imaginá-la é como uma **única coluna de uma planilha** ou uma lista Python com superpoderes.

Uma `Series` é um objeto unidimensional, semelhante a um vetor, que pode armazenar qualquer tipo de dado (números inteiros, floats, strings, etc.). Sua principal característica, que a diferencia de uma simples lista, é a presença de um **índice (index)**. O índice é um conjunto de rótulos que nos permite acessar cada elemento da `Series` de forma rápida e intuitiva.

Caso Prático

Imagine que uma equipe de pesquisadores do Instituto Nacional de Pesquisas da Amazônia (INPA) contou a quantidade de espécies de peixes em três lagos diferentes perto de Manaus. Podemos representar esses dados em uma `Series`.

Copie e Teste!

```
import pandas as pd

# Criando uma Series a partir de uma lista Python
dados_peixes = [120, 85, 98]
lagos = ["Lago do Puraquequara", "Lago do Janauari", "Lago do
   Tarumã-Açu"]

# Criando a Series com índices personalizados
s_peixes = pd.Series(data=dados_peixes, index=lagos)

print("Dados da Contagem de Espécies de Peixes:")
print(s_peixes)
```

Saída Esperada

```
Dados da Contagem de Espécies de Peixes:
Lago do Puraquequara    120
Lago do Janauari         85
Lago do Tarumã-Açu       98
dtype: int64
```

Observe a saída: não é apenas uma lista de números. À esquerda, temos os **rótulos do índice** (os nomes dos lagos) e, à direita, os **valores** correspondentes. No final, `dtype: int64` nos informa que o Pandas identificou os dados como números inteiros. Essa estrutura de índice + valor é o que torna a `Series` tão poderosa para a análise de dados.

Fique Alerta!

O índice é o coração da `Series`. Enquanto em uma lista Python você acessa os elementos apenas por sua posição numérica (0, 1, 2...), em uma `Series` você pode usar rótulos explícitos, como nomes de locais ou datas. Isso torna o código muito mais legível e menos propenso a erros.

O `DataFrame`: A Tabela Superpoderosa

Se a `Series` é uma única coluna, o **`DataFrame`** é a tabela inteira. É a estrutura de dados mais importante e utilizada do Pandas, projetada para representar dados tabulares com colunas de diferentes tipos, exatamente como uma planilha do Excel ou uma tabela de um banco de dados.

A melhor maneira de entender um `DataFrame` é imaginá-lo como um **dicionário de `Series`**. Cada coluna do `DataFrame` é, na verdade, uma `Series`, e todas essas `Series` (colunas) compartilham o mesmo índice (as linhas). Essa estrutura bidimensional (linhas e colunas) é perfeita para a maioria dos conjuntos de dados que encontramos no mundo real.

Caso Prático

Buscando sofisticar o caso anterior equipe de pesquisa não anotou apenas a contagem de peixes, mas também o nível de pH e a turbidez da água em cada um dos lagos. Agora temos um conjunto de dados tabular, perfeito para um `DataFrame`.

Copie e Teste!

```
import pandas as pd

# Dicionário com os dados da pesquisa
dados_lagos = {
    'especies_peixes': [120, 85, 98],
    'ph_agua': [6.5, 6.8, 6.2],
    'turbidez_ntu': [4.5, 3.1, 5.8]
}

# Lista com os nomes dos lagos para usar como índice
nomes_lagos = ["Lago do Puraquequara", "Lago do Janauari", "Lago
   do Tarumã-Açu"]

# Criando o DataFrame
df_lagos = pd.DataFrame(dados_lagos, index=nomes_lagos)

print("Tabela de Monitoramento da Qualidade da Água e
   Biodiversidade:")
print(df_lagos)
```

Saída Esperada

```
Tabela de Monitoramento da Qualidade da Água e
   Biodiversidade:
                     especies_peixes  ph_agua  turbidez_ntu
Lago do Puraquequara              120      6.5           4.5
Lago do Janauari                   85      6.8           3.1
Lago do Tarumã-Açu                 98      6.2           5.8
```

Veja como o Pandas organizou nossos dados em uma tabela limpa e bem formatada:

- **Índice (Index):** Os rótulos das linhas à esquerda (os nomes dos lagos).
- **Colunas (Columns):** Os rótulos no topo, que vieram das chaves do nosso dicionário.
- **Dados (Data):** Os valores que preenchem a tabela.

A genialidade do `DataFrame` está em como ele conecta tudo. Se quisermos analisar apenas a turbidez da água, podemos selecionar essa coluna, e o que o Pandas nos devolve? Uma `Series`!

Copie e Teste!

```
# Criando o dataframe de forma compacta
dados_lagos = {'especies_peixes': [120, 85, 98], 'ph_agua': [6.5,
   6.8, 6.2], 'turbidez_ntu': [4.5, 3.1, 5.8]}
nomes_lagos = ["Lago do Puraquequara", "Lago do Janauari", "Lago
   do Tarumã-Açu"]
df_lagos = pd.DataFrame(dados_lagos, index=nomes_lagos)

# Selecionando uma única coluna do DataFrame
coluna_ph = df_lagos['ph_agua']

print("A coluna 'ph_agua' é do tipo:")
print(type(coluna_ph))
print("\nConteúdo da coluna:")
print(coluna_ph)
```

Saída Esperada

```
A coluna 'ph_agua' é do tipo:
<class 'pandas.core.series.Series'>

Conteúdo da coluna:
Lago do Puraquequara    6.5
Lago do Janauari        6.8
Lago do Tarumã-Açu      6.2
Name: ph_agua, dtype: float64
```

Este exemplo prático ilustra a relação fundamental entre as duas estruturas aboradadas. Essencialemnte, um `DataFrame` é um contêiner que organiza múltiplas `Series` que compartilham um mesmo índice. Dominar essa ideia, como enfatizado pelo próprio criador do Pandas, Wes McKinney, é crucial para a manipulação eficaz de dados em Python (McKinney, 2018).

Agora que sabemos o que são e como criar as estruturas de `Series` e `DataFrame`, estamos prontos para o próximo passo: aprender a carregar dados diretamente de arquivos, que é a forma mais comum de iniciar qualquer projeto de análise de dados.

3.1.3 Lendo Dados de Arquivos com Pandas

Nos tópicos anteriores, aprendemos a criar DataFrames e Series, que são as estruturas fundamentais do Pandas. Vimos que um DataFrame é como uma tabela superpoderosa, pronta para a análise. Mas, na maioria das vezes, nossos dados não nascem dentro do nosso código, eles vivem em arquivos no mundo exterior.

A função `pd.read_csv()` é a principal porta de entrada para trazer dados tabulares para o Pandas. Dominar essa etapa é o que transforma seus arquivos estáticos em DataFrames dinâmicos e interativos, prontos para serem explorados.

Caso Prático

Uma cooperativa de produtores de açaí da região amazônica registrou a produção de cada mês em um arquivo chamado `producao_acai.csv`. Nossa primeira tarefa é carregar esses dados em um DataFrame para podermos começar a nossa análise.

Saída Esperada

```
mes,producao_toneladas,local
Janeiro,15.5,Amapá
Fevereiro,12.8,Pará
Março,18.2,Amapá
Abril,17.9,Pará
```

Copie e Teste!

```python
import pandas as pd

# Usamos a função read_csv para ler o arquivo
# O Pandas identifica automaticamente o cabeçalho e as colunas
df_acai = pd.read_csv('producao_acai.csv')

# Vamos imprimir o DataFrame para ver o resultado
print("Dados de Produção de Açaí:")
print(df_acai)
```

Saída Esperada

```
Dados de Produção de Açaí:
         mes  producao_toneladas    local
0    Janeiro                15.5    Amapá
1  Fevereiro                12.8     Pará
2      Março                18.2    Amapá
3      Abril                17.9     Pará
```

Com uma única linha de código, `pd.read_csv()`, o Pandas realizou todo o trabalho pesado de abrir o arquivo, interpretar sua estrutura e carregar os dados em um DataFrame limpo e organizado. Essa simplicidade e poder são o que fazem do Pandas a ferramenta preferida para análise de dados em Python (McKinney, 2018).

Ajustando a Leitura com Parâmetros

O mundo real é bagunçado, e os dados também. Nem todo arquivo CSV usa vírgulas como separador ou tem uma linha de cabeçalho. Felizmente, a função `read_csv()` é extremamente flexível e possui dezenas de parâmetros para lidar com essas variações. Vamos explorar os mais comuns.

Caso Prático

Um sistema de monitoramento mais antigo, que mede a qualidade do solo, gera arquivos de dados usando o ponto e vírgula (`;`) como separador. Precisamos ler o arquivo `dados_solo.csv` corretamente.

Saída Esperada

```
ponto_coleta;ph;umidade
Ponto_A;5.8;0.45
Ponto_B;6.2;0.51
```

Para isso, usamos o parâmetro `delimiter` (ou `sep`) para informar ao Pandas qual caractere está separando as colunas.

Copie e Teste!

```
import pandas as pd

# Especificamos que o delimitador é um ponto e vírgula
df_solo = pd.read_csv('dados_solo.csv', delimiter=';')

print("Dados de Qualidade do Solo:")
print(df_solo)
```

Às vezes, um arquivo pode não ter uma linha de cabeçalho. Nesse caso podemos nomear as colunas manualmente ou simplesmente usar `header=None`. Podemos ainda redefinir uma das colunas do arquivo como índice do DataFrame através do parâmetro `index_col`

Salvando Dados em Arquivos

A análise de dados é um ciclo: lemos os dados, limpamos, transformamos, analisamos e, finalmente, salvamos nossos resultados. Salvar nosso trabalho é crucial para compartilhar nossas descobertas, criar relatórios ou usar os dados processados em outra análise. O Pandas torna esse processo tão fácil quanto a leitura, com os métodos `to_csv()` e `to_excel()`.

Caso Prático

Após analisar os dados de produção de açaí, calculamos a produção média por estado e queremos salvar esse novo resumo em um arquivo para um relatório.

Copie e Teste!

```
import pandas as pd

df_acai = pd.read_csv('producao_acai.csv')

# Agrupamos os dados por local e calculamos a média da produção
resumo_producao = df_acai.groupby('local')['producao_toneladas'].
   mean().reset_index()

print("Resumo a ser salvo:")
print(resumo_producao)

# Salvando o DataFrame em um novo arquivo CSV
resumo_producao.to_csv('resumo_acai_por_estado.csv')

print("\nArquivo 'resumo_acai_por_estado.csv' salvo com sucesso!")
```

Fique Alerta!

Se você abrir o arquivo `resumo_acai_por_estado.csv`, verá que o Pandas salvou os dados, mas incluiu uma coluna extra com os índices do DataFrame (0, 1). Na maioria das vezes, não queremos isso. Por padrão, o método `to_csv()` salva o índice do DataFrame como a primeira coluna no arquivo. Para evitar isso, que é o comportamento desejado na maioria dos casos, use o parâmetro `index=False`.

```
# Salvando sem o índice
resumo_producao.to_csv('resumo_acai_por_estado.csv', index=False)
```

Conhecendo um pouco mais!

Além de arquivos CSV, o Pandas pode ler e escrever em muitos outros formatos. Um dos mais comuns é o Excel.

- Para ler um arquivo Excel: `df = pd.read_excel('meu_arquivo.xlsx')`
- Para salvar em um arquivo Excel: `df.to_excel('meu_resultado.xlsx', index=False)`

Dica: Para trabalhar com arquivos Excel, talvez você precise instalar uma biblioteca adicional. Abra seu terminal e digite:

```
pip install openpyxl
```

Saber ler e escrever dados de e para arquivos é a espinha dorsal de qualquer fluxo de trabalho de análise de dados. Com o Pandas, essas tarefas complexas se tornam simples e eficientes, liberando seu tempo para a parte mais importante: extrair insights e contar as histórias que seus dados escondem.

3.2 Inspeção e Manipulação de Dados

No tópico anterior, você abriu a porta para o mundo da análise de dados ao carregar seu primeiro arquivo em um DataFrame. Foi um passo gigantesco! Agora que os dados estão ”dentro de casa”, o que fazemos com eles? É como um detetive que acaba de receber uma caixa de evidências: o primeiro passo não é tirar conclusões, mas sim inspecionar cuidadosamente cada item. Esta seção é sobre isso: aprender a investigar, organizar e preparar seus dados para que eles possam, de fato, contar uma história confiável.

Carregar um arquivo CSV em um DataFrame com `pd.read_csv()` é como retornar de uma expedição na Amazônia com todos os seus cadernos de campo e amostras. Todo o material bruto está sobre a mesa, pronto para ser analisado. No entanto, o trabalho de um pesquisador está longe de terminar; na verdade, ele está apenas começando.

Os dados do mundo real, assim como as anotações de campo, raramente são perfeitos. Você pode encontrar medições faltando porque um sensor falhou, anotações inconsistentes feitas em dias diferentes, valores que parecem completamente fora do padrão ou simplesmente uma grande quantidade de informações onde você só precisa de uma pequena parte. Tentar fazer uma análise ou criar um gráfico com esses dados ”sujos”é como tentar construir um mapa preciso a partir de anotações borradas e páginas rasgadas: o resultado final não será confiável.

Esta seção é dedicada à etapa mais crucial de qualquer projeto de análise de dados: a **inspeção e manipulação**. Antes de podermos extrair *insights*, precisamos garantir que nossos dados estejam limpos, organizados e no formato correto. É o processo de arregaçar as mangas e transformar o material bruto em uma fonte de informações preparada e confiável. Para isso, vamos aprender a usar as ferramentas que o Pandas nos oferece para:

- **Dar os primeiros passos na inspeção:** Vamos aprender a ”dar uma primeira olhada”no nosso DataFrame, entendendo suas dimensões, os tipos de dados que ele contém e visualizando algumas linhas para ter uma noção geral do nosso conjunto de dados.
- **Selecionar e filtrar informações:** Assim como um biólogo foca em uma espécie específica ou um geógrafo em uma determinada região, aprenderemos a usar uma ”lupa”para selecionar apenas as linhas e colunas que nos interessam para uma análise específica.
- **Lidar com dados ausentes:** O que fazer com as ”anotações em branco”? Investigaremos estratégias para encontrar, analisar e tratar os valores nulos ou faltantes, uma das tarefas mais comuns e importantes na limpeza de dados.
- **Modificar e transformar o DataFrame:** Por fim, veremos como podemos corrigir informações, criar novas colunas a partir de dados existentes e reorganizar nossa tabela para que ela se ajuste perfeitamente às nossas necessidades de análise.

Dominar estas técnicas é o que diferencia uma análise superficial de uma investigação de dados profunda e precisa. Uma parte significativa do trabalho de um analista de dados é gasta justamente na preparação e limpeza dos dados, pois é essa base sólida que permite a descoberta de conhecimento verdadeiro e a criação de visualizações impactantes (McKinney, 2018). Vamos começar a construir essa base.

3.2.1 Primeiros Passos: Inspecionando seu DataFrame

No tópico anterior, você abriu a porta para o mundo da análise de dados ao carregar seu primeiro arquivo em um DataFrame. Foi um passo gigantesco! Agora que os dados estão ”dentro de casa”, o que fazemos com eles? É como um detetive que acaba de receber uma caixa de evidências: o primeiro passo não é tirar conclusões, mas sim inspecionar cuidadosamente cada item para entender com o que está lidando.

Esta etapa de inspeção inicial é, talvez, a mais importante de toda a sua jornada. Antes de calcular médias, encontrar o maior valor ou criar um gráfico, você precisa ”sentir”seus dados. Quantas informações você tem? Elas estão completas? O Python as entendeu da maneira correta? Responder a essas perguntas evita erros e garante que sua análise seja construída sobre uma base sólida e confiável.

Figura 3.3: Arte que ilustra o conceito de inspeção de dados
Fonte: Gerada por Google Gemini 2.5 Pro, 2025

Felizmente, o Pandas nos oferece um kit de ferramentas de investigação de primeira linha. Com apenas alguns comandos simples, podemos obter um resumo completo do nosso conjunto de dados. Vamos aprender a usar as cinco ferramentas essenciais de todo analista de dados: `.head()`, `.tail()`, `.shape`, `.columns` e, a mais poderosa de todas, `.info()`.

Caso Prático

Monitoramento de Peixes no Rio Amazonas
Imagine que uma equipe de pesquisadores do Instituto Nacional de Pesquisas da Amazônia (INPA) coletou dados sobre peixes em diferentes pontos ao longo do Rio Amazonas. Eles registraram o local, a espécie do peixe, a quantidade de peixes contados e o pH da água. Os dados foram salvos no arquivo `coleta_peixes.csv`.

Saída Esperada

Conteúdo de coleta_peixes.csv

```
local,especie,quantidade,ph_agua
Encontro das Águas,Tambaqui,45,6.8
Frente a Manaus,Tucunaré,22,6.5
Lago Janauari,Pirarucu,10,7.1
Parintins,Tambaqui,56,
Frente a Manaus,Jaraqui,89,6.6
Lago Janauari,Tucunaré,15,7.0
Encontro das Águas,Pirarucu,,6.9
```

Após dar uma observada geral nos dados, nosso primeiro passo em qualquer análise é carregar esses dados em um DataFrame do Pandas.

Copie e Teste!

```python
import pandas as pd

# Carregando os dados do arquivo CSV para o nosso DataFrame
df_peixes = pd.read_csv('coleta_peixes.csv')

# Exibindo o DataFrame completo para confirmar que carregou
print(df_peixes)
```

Saída Esperada

```
                local   especie  quantidade  ph_agua
0  Encontro das Águas  Tambaqui        45.0      6.8
1     Frente a Manaus  Tucunaré        22.0      6.5
2       Lago Janauari  Pirarucu        10.0      7.1
3           Parintins  Tambaqui        56.0      NaN
4     Frente a Manaus   Jaraqui        89.0      6.6
5       Lago Janauari  Tucunaré        15.0      7.0
6  Encontro das Águas  Pirarucu         NaN      6.9
```

Dando uma Espiada: `.head()` e `.tail()`

Quando lidamos com arquivos que podem ter milhares ou milhões de linhas, imprimir o DataFrame inteiro é impraticável. A primeira coisa que um analista de dados faz é "espiar"o começo e o fim do seu conjunto de dados.

- O método **`.head()`** mostra as primeiras 5 linhas por padrão. É como olhar os primeiros itens que estão no topo da caixa de evidências.
- O método **`.tail()`** mostra as últimas 5 linhas. É útil para ver se o final do arquivo foi lido corretamente.

Copie e Teste!

```
# Mostrando as 3 primeiras linhas
print("--- Primeiras 3 linhas com .head(3) ---")
print(df_peixes.head(3))

# Mostrando as 2 últimas linhas
print("\n--- Últimas 2 linhas com .tail(2) ---")
print(df_peixes.tail(2))
```

Saída Esperada

```
--- Primeiras 3 linhas com .head(3) ---
                local   especie  quantidade  ph_agua
0  Encontro das Águas  Tambaqui        45.0      6.8
1     Frente a Manaus  Tucunaré        22.0      6.5
2       Lago Janauari  Pirarucu        10.0      7.1

--- Últimas 2 linhas com .tail(2) ---
                local   especie  quantidade  ph_agua
5       Lago Janauari  Tucunaré        15.0      7.0
6  Encontro das Águas  Pirarucu         NaN      6.9
```

Essa primeira olhada já nos dá uma boa ideia da estrutura dos dados e até revela algo interessante no final: um valor `NaN` (Not a Number), que é como o Pandas representa dados ausentes. Já encontramos nossa primeira pista!

Qual o Tamanho do Desafio? Conhecendo as Dimensões com `.shape`

A próxima pergunta é: "Com quantos dados estamos lidando?". O atributo **`.shape`** nos dá a resposta exata. Ele não é um método (por isso não usa parênteses `()`), mas sim uma propriedade do DataFrame que retorna uma tupla com o formato `(número de linhas, número de colunas)`.

Copie e Teste!

```
# Obtendo as dimensões do nosso DataFrame
dimensoes = df_peixes.shape

print(f"O DataFrame tem {dimensoes[0]} linhas (observações) e {
    dimensoes[1]} colunas (variáveis).")
```

Saída Esperada

```
O DataFrame tem 7 linhas (observações) e 4 colunas (
    variáveis).
```

Saber o `.shape` é fundamental para entender a escala do seu problema. Sete linhas é um conjunto pequeno, mas em um projeto real, você poderia ver `(700000, 15)`, indicando um desafio muito maior.

A Ficha Técnica Completa: O Poder do `.info()`

Se `.head()` é uma espiada e `.shape` mede o tamanho, o método **`.info()`** é o raio-X completo do seu DataFrame. Ele fornece um resumo técnico conciso e extremamente útil, sendo o comando mais importante para a inspeção inicial.

Copie e Teste!

```
# Obtendo um resumo completo do DataFrame
print("--- Resumo técnico do DataFrame com .info() ---")
df_peixes.info()
```

Saída Esperada

```
--- Resumo técnico do DataFrame com .info() ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   local       7 non-null      object
 1   especie     7 non-null      object
 2   quantidade  6 non-null      float64
 3   ph_agua     6 non-null      float64
dtypes: float64(2), object(2)
memory usage: 356.0+ bytes
```

Esta saída é uma mina de ouro de informações. Vamos analisá-la parte por parte:

1. **`RangeIndex: 7 entries, 0 to 6`**: Confirma que temos 7 linhas, com índices que vão de 0 a 6.
2. **`Data columns (total 4 columns)`**: Confirma que temos 4 colunas.
3. **A Tabela de Colunas**: Esta é a parte mais importante.
   - `Column`: O nome de cada coluna.
   - `Non-Null Count`: O número de valores que **não são nulos**. Esta é a nossa principal ferramenta para detectar dados ausentes! Observe que as respectivas colunas `quantidade` e `ph_agua` possuem "6 non-null", enquanto o total de linhas é 7. Isso confirma que cada uma delas tem um valor faltando.
   - `Dtype`: O tipo de dado que o Pandas atribuiu a cada coluna. `object` geralmente significa que a coluna contém strings (texto). `float64` indica que a coluna contém números com casas decimais.

Fique Alerta!

O método `.info()` é seu melhor amigo no início de qualquer análise. Ele responde a três perguntas cruciais de uma só vez:

1. Quais são as minhas colunas?
2. Existem dados faltando? (Compare "Non-Null Count"com o total de "entries")
3. Os tipos de dados estão corretos? (Números as vezes podem ser lidos como texto)

Com esses comandos, você realizou uma inspeção inicial completa. Em poucos minutos, você entendeu o tamanho, a estrutura, os tipos de dados e até identificou problemas de dados ausentes em seu conjunto de dados. Como enfatiza Wes McKinney, o criador do Pandas, dedicar tempo para entender a estrutura dos seus dados no início economiza um tempo imenso e evita erros no futuro (McKinney, 2018).

Agora que sabemos com o que estamos lidando, estamos prontos para o próximo passo: aprender a focar em partes específicas dos nossos dados, selecionando e filtrando as informações que são mais relevantes para nossa investigação.

3.2.2 Seleção e Filtragem de Dados

No tópico anterior, você aprendeu a carregar seus dados e a realizar uma primeira inspeção, como um detetive examinando a caixa de evidências. Agora que temos uma visão geral do nosso `DataFrame`, é hora de aprofundar a investigação. Raramente trabalhamos com todos os dados de uma vez. Na maioria das vezes, estamos interessados em responder a perguntas específicas, como:

- "Qual a quantidade de Pirarucus avistados no Lago Janauari?"
- "Quais locais apresentaram um pH da água abaixo do nível ideal de 6.5?"
- "Quais são os dados apenas das colunas `especie` e `local`?"

Para responder a essas e outras perguntas, precisamos aprender a "fatiar"e "peneirar"o `DataFrame`, selecionando apenas as partes que nos interessam. Esta seção é dedicada a ensinar você a usar as ferramentas de seleção e filtragem do Pandas. Pense nelas como a lupa e as pinças do detetive: ferramentas de precisão para focar nos detalhes mais importantes e isolar as evidências cruciais.

Dominar a seleção e a filtragem é o que transforma a análise de dados de uma simples observação para uma investigação direcionada, permitindo que você extraia respostas claras de conjuntos de dados complexos.

A Lupa do Detetive: Selecionando Colunas

A tarefa mais comum em uma análise é focar em uma ou mais variáveis (colunas) específicas. Em nosso `DataFrame` de peixes, por exemplo, podemos querer analisar apenas as espécies, ignorando temporariamente as medições de pH. O Pandas torna essa tarefa extremamente simples e intuitiva.

Selecionando uma Única Coluna: Para selecionar uma única coluna, basta usar colchetes `[]` com o nome da coluna entre aspas, como se estivéssemos acessando um valor em um dicionário.

Caso Prático

Vamos continuar nossa análise com os dados de monitoramento de peixes. Nossa primeira tarefa é isolar e visualizar apenas a coluna que contém as espécies coletadas.

Copie e Teste!

```
import pandas as pd

# Recriando nosso DataFrame para o exemplo
dados = {
    'local': ['Encontro das Águas', 'Frente a Manaus', 'Lago
   Janauari', 'Parintins', 'Frente a Manaus', 'Lago Janauari', '
   Encontro das Águas'],
    'especie': ['Tambaqui', 'Tucunaré', 'Pirarucu', 'Tambaqui', '
   Jaraqui', 'Tucunaré', 'Pirarucu'],
    'quantidade': [45.0, 22.0, 10.0, 56.0, 89.0, 15.0, None],
    'ph_agua': [6.8, 6.5, 7.1, None, 6.6, 7.0, 6.9]
}
df_peixes = pd.DataFrame(dados)

# Selecionando a coluna 'especie'
coluna_especies = df_peixes['especie']

print("--- Conteúdo da Coluna 'especie' ---")
print(coluna_especies)

print("\nTipo do objeto retornado:")
print(type(coluna_especies))
```

Saída Esperada

```
--- Conteúdo da Coluna 'especie' ---
0    Tambaqui
1    Tucunaré
2    Pirarucu
3    Tambaqui
4     Jaraqui
5    Tucunaré
6    Pirarucu
Name: especie, dtype: object

Tipo do objeto retornado:
<class 'pandas.core.series.Series'>
```

Fique Alerta!

Observe que, ao selecionar uma única coluna, o Pandas não nos devolve um `DataFrame`, mas sim uma **`Series`**. Isso reforça a ideia de que um `DataFrame` é, essencialmente, uma coleção de `Series` que compartilham o mesmo índice.

Selecionando Múltiplas Colunas Para selecionar várias colunas ao mesmo tempo, como `local` e `quantidade`, utilizamos uma **lista de nomes de colunas** dentro dos colchetes.

Copie e Teste!

```
# Para selecionar múltiplas colunas, usamos uma lista de nomes
colunas_selecionadas = df_peixes[['local', 'quantidade']]

print("--- DataFrame com as colunas 'local' e 'quantidade' ---")
print(colunas_selecionadas)

print("\nTipo do objeto retornado:")
print(type(colunas_selecionadas))
```

Saída Esperada

```
--- DataFrame com as colunas 'local' e 'quantidade' ---
                 local  quantidade
0  Encontro das Águas        45.0
1     Frente a Manaus        22.0
2       Lago Janauari        10.0
3           Parintins        56.0
4     Frente a Manaus        89.0
5       Lago Janauari        15.0
6  Encontro das Águas         NaN

Tipo do objeto retornado:
<class 'pandas.core.frame.DataFrame'>
```

Desta vez, como selecionamos mais de uma coluna, o resultado é um novo `DataFrame`, menor e mais focado. Essa técnica é fundamental para criar visualizações ou análises que dependem apenas de um subconjunto das suas variáveis.

Pinças de Precisão: Selecionando Linhas com `.loc` e `.iloc`

O Pandas nos oferece duas ”pinças”para essa tarefa, , cada uma com sua especialidade:

1. **`.loc` (Seleção por Rótulo)**: Utiliza os **nomes** (rótulos) do índice para encontrar as linhas. É como encontrar um livro na estante pelo seu título.

2. **`.iloc` (Seleção por Posição Inteira)**: Utiliza a **posição numérica** (índice inteiro) das linhas, começando do 0. É como pegar o ”terceiro livro da prateleira de cima”, independentemente do seu título.

Seleção por Rótulo com `.loc` O método `.loc` é a principal forma de selecionar dados quando os rótulos do seu índice são significativos (como datas, nomes de locais, etc.). No nosso `df_peixes`, o índice padrão é numérico (0, 1, 2...), então os rótulos são os próprios números.

Caso Prático

Vamos refazer nosso `DataFrame` de monitoramento de lagos, desta vez usando os nomes dos lagos como índice, e depois usar o `.loc` para selecionar os dados de um lago específico.

Copie e Teste!

```
import pandas as pd

dados_lagos = {
    'especies_peixes': [120, 85, 98],
    'ph_agua': [6.5, 6.8, 6.2],
    'turbidez_ntu': [4.5, 3.1, 5.8]
}
nomes_lagos = ["Lago do Puraquequara", "Lago do Janauari", "Lago
   do Tarumã-Açu"]
df_lagos = pd.DataFrame(dados_lagos, index=nomes_lagos)

print("--- DataFrame com Índice Nominal ---")
print(df_lagos)

# Usando .loc para selecionar a linha com o rótulo "Lago do
   Janauari"
dados_janauari = df_lagos.loc["Lago do Janauari"]

print("\n--- Dados apenas do Lago do Janauari ---")
print(dados_janauari)
```

Saída Esperada

```
--- DataFrame com Índice Nominal ---
                     especies_peixes  ph_agua  turbidez_ntu
Lago do Puraquequara              120      6.5           4.5
Lago do Janauari                   85      6.8           3.1
Lago do Tarumã-Açu                 98      6.2           5.8

--- Dados apenas do Lago do Janauari ---
especies_peixes    85.0
ph_agua             6.8
turbidez_ntu        3.1
Name: Lago do Janauari, dtype: float64
```

Seleção por Posição com `.iloc` O método `.iloc` funciona de forma similar, mas ignora completamente os rótulos e foca apenas na posição numérica. É extremamente útil quando

você não conhece os rótulos do índice ou quer simplesmente pegar ”a primeira linha”ou ”as últimas cinco linhas”.

Copie e Teste!

```
# Selecionando a primeira linha (posição 0) do df_peixes
primeira_linha = df_lagos.iloc[0]
print("--- Primeira Linha do DataFrame (posição 0) ---")
print(primeira_linha)

# Selecionando a terceira linha (posição 2)
terceira_linha = df_lagos.iloc[2]
print("\n--- Terceira Linha do DataFrame (posição 2) ---")
print(terceira_linha)

# Selecionando um intervalo de linhas (da posição 1 à 2)
intervalo_linhas = df_lagos.iloc[1:4]
print("\n--- Linhas da Posição 1 até a 2 ---")
print(intervalo_linhas)
```

Saída Esperada

```
--- Primeira Linha do DataFrame (posição 0) ---
especies_peixes    120.0
ph_agua              6.5
turbidez_ntu         4.5
Name: Lago do Puraquequara, dtype: float64

--- Terceira Linha do DataFrame (posição 2) ---
especies_peixes    98.0
ph_agua             6.2
turbidez_ntu        5.8
Name: Lago do Tarumã-Açu, dtype: float64

--- Linhas da Posição 1 até a 3 ---
                     especies_peixes  ph_agua  turbidez_ntu
1 Lago do Janauari                85      6.8           3.1
2 Lago do Tarumã-Açu              98      6.2           5.8
```

Observe que o dataframe não possuí a posição 4, portanto, ele executa somente até a posição 2 que se refere ao índice máximo que ele alcança.

A Peneira Fina: Filtragem por Condições

A seleção de colunas e linhas é poderosa, mas a verdadeira mágica da análise de dados acontece quando começamos a **filtrar** nossos dados com base em condições. É aqui que passamos de ”me mostre esta linha”para ”me mostre todas as linhas **onde** algo interessante acontece”.

Essa técnica, conhecida como **indexação booleana** ou **máscara booleana**, funciona em dois passos:

1. **Criar uma condição**: Fazemos uma pergunta sobre uma coluna que resulta em `True` ou `False` para cada linha. (Ex: `df_peixes['quantidade'] > 50`).
2. **Aplicar a máscara**: Usamos essa série de `True`/`False` para filtrar o `DataFrame`, que nos retorna apenas as linhas onde a condição foi `True`.

Filtragem com uma Única Condição A filtragem por condição única envolve aplicar uma regra lógica (como > ou ==) a uma coluna. Essa operação gera uma "máscara" de valores `True` e `False`, que é usada para selecionar do `DataFrame` apenas as linhas onde a condição é verdadeira.

Caso Prático

A equipe de pesquisa quer identificar todas as coletas onde a contagem de peixes foi alta, definida como mais de 50 indivíduos. Precisamos filtrar nosso `DataFrame` para mostrar apenas esses registros.

Copie e Teste!

```
# Passo 1: Criar a máscara booleana
# Isso cria uma Series de True/False
mascara_alta_quantidade = df_peixes['quantidade'] > 50
print("--- Máscara Booleana (Quantidade > 50) ---")
print(mascara_alta_quantidade)

# Passo 2: Aplicar a máscara ao DataFrame
coletas_grandes = df_peixes[mascara_alta_quantidade]
print("\n--- Coletas com mais de 50 indivíduos ---")
print(coletas_grandes)
```

Saída Esperada

```
--- Máscara Booleana (Quantidade > 50) ---
0    False
1    False
2    False
3     True
4     True
5    False
6    False
Name: quantidade, dtype: bool

--- Coletas com mais de 50 indivíduos ---
             local  especie  quantidade  ph_agua
3        Parintins  Tambaqui        56.0      NaN
4  Frente a Manaus   Jaraqui        89.0      6.6
```

Filtragem com Múltiplas Condições E se precisarmos de critérios mais complexos? Por exemplo, encontrar os registros de "Tambaqui"**E** onde a quantidade foi maior que 50. Para combinar múltiplas condições, usamos os operadores lógicos:

- `&` para **E** (ambas as condições devem ser `True`)
- `|` para **OU** (pelo menos uma das condições deve ser `True`)

Fique Alerta!

Ao combinar múltiplas condições no Pandas, é obrigatório envolver cada condição individual em parênteses `()`. Isso ocorre devido à forma como o Python processa as operações, e esquecer os parênteses resultará em um erro.

Caso Prático

Os pesquisadores agora querem encontrar registros de monitoramento que são particularmente preocupantes: locais onde o pH da água é considerado ácido (menor que 6.7) **OU** onde a espécie registrada é o Pirarucu (uma espécie importante para a conservação).

Copie e Teste!

```
# Criando as duas máscaras separadamente
mascara_ph_acido = df_peixes['ph_agua'] < 6.7
mascara_pirarucu = df_peixes['especie'] == 'Pirarucu'

# Combinando as máscaras com o operador OU (|)
registros_importantes = df_peixes[mascara_ph_acido |
   mascara_pirarucu]

print("--- Registros com pH ácido OU da espécie Pirarucu ---")
print(registros_importantes)
```

Saída Esperada

```
--- Registros com pH ácido OU da espécie Pirarucu ---
                local   especie  quantidade  ph_agua
1     Frente a Manaus  Tucunaré        22.0      6.5
2       Lago Janauari  Pirarucu        10.0      7.1
4     Frente a Manaus   Jaraqui        89.0      6.6
6  Encontro das Águas  Pirarucu         NaN      6.9
```

Ao dominar a seleção e a filtragem, você ganhou o poder de fazer perguntas complexas aos seus dados e obter respostas precisas. Essa é a base de toda a análise exploratória de dados. Agora que sabemos como isolar as informações que nos interessam, o próximo passo é aprender a lidar com um dos problemas mais comuns em conjuntos de dados do mundo real: as informações que estão faltando.

3.2.3 Lidando com Dados Ausentes (Missing Data)

Nos tópicos anteriores, você aprendeu a usar suas ferramentas de detetive para inspecionar e fatiar o DataFrame. Você já sabe como olhar as primeiras linhas, verificar as dimensões e filtrar os dados que lhe interessam. Contudo, ao fazer essa investigação, você inevitavelmente encontrará um dos desafios mais comuns na análise de dados do mundo real: as lacunas na informação.

Imagine que um pesquisador, ao catalogar espécies em uma área remota da Amazônia, anota suas observações em um caderno. Em um dia de chuva forte, a tinta de uma das anotações borra, tornando a contagem de uma espécie ilegível. Em outro momento, a bateria do GPS acaba, e ele não consegue registrar as coordenadas de uma planta rara. Ao final da expedição, seu caderno, assim como nossos DataFrames, terá espaços em branco — dados ausentes.

Esses dados faltantes, representados no Pandas pelo valor especial `NaN` (Not a Number), não são apenas espaços vazios; são perguntas sem resposta. Tentar calcular uma média de uma coluna com valores `NaN` sem o devido cuidado pode levar a erros ou a resultados incorretos, comprometendo toda a sua análise.

Nesta seção, vamos aprender as técnicas essenciais para lidar com esses ”buracos”em nossos dados. Você aprenderá a ser um verdadeiro ”restaurador”de informações, dominando as duas estratégias principais:

- **Detecção:** Como encontrar e quantificar sistematicamente os dados faltantes presentes em seu DataFrame.
- **Tratamento:** O que fazer após encontrar os dados ausentes. Veremos as duas abordagens principais:
    1. **Remoção:** Quando é seguro remover as linhas ou colunas com dados faltantes.
    2. **Preenchimento (Imputação):** Como preencher as lacunas de forma inteligente, usando valores estatísticos ou lógicos para preservar a integridade do seu conjunto de dados.

Dominar a arte de tratar dados ausentes é uma das habilidades mais importantes de um analista de dados, pois garante que as histórias que você conta com seus dados sejam precisas, confiáveis e completas.

Detecção: Onde Estão as Lacunas?

Antes de podermos consertar um problema, precisamos saber onde ele está e qual a sua dimensão. No tópico anterior, o método `.info()` já nos deu uma primeira pista ao mostrar a contagem de valores ”non-null”(não nulos). Agora, vamos usar uma ferramenta mais direta para contar exatamente quantos `NaN`s existem em cada coluna.

A combinação dos métodos `.isnull()` e `.sum()` é a maneira mais eficiente de realizar essa tarefa.

1. `.isnull()`: Este método percorre todo o DataFrame e retorna um novo DataFrame do mesmo tamanho, mas preenchido com valores booleanos: `True` onde havia um `NaN` e `False` nos demais locais.
2. `.sum()`: Quando aplicado a um DataFrame booleano, este método trata `True` como `1` e `False` como `0`. O resultado é a soma de todos os `1`s em cada coluna, ou seja, a contagem exata de dados ausentes.

Caso Prático

Vamos continuar com nosso DataFrame de coleta de peixes, que já sabemos ter alguns dados faltando. Nossa tarefa é usar a combinação `.isnull().sum()` para obter um relatório claro da quantidade de dados ausentes em cada coluna.

Copie e Teste!

```
import pandas as pd
import numpy as np # Usaremos numpy para inserir NaN de forma
   controlada

# Recriando nosso DataFrame para o exemplo
dados = {
    'local': ['Encontro das Águas', 'Frente a Manaus', 'Lago
   Janauari', 'Parintins', 'Frente a Manaus', 'Lago Janauari', '
   Encontro das Águas'],
    'especie': ['Tambaqui', 'Tucunaré', 'Pirarucu', 'Tambaqui', '
   Jaraqui', 'Tucunaré', 'Pirarucu'],
    'quantidade': [45.0, 22.0, 10.0, 56.0, 89.0, 15.0, np.nan],
    'ph_agua': [6.8, 6.5, 7.1, np.nan, 6.6, 7.0, 6.9]
}
df_peixes = pd.DataFrame(dados)

# Usando .isnull().sum() para contar os dados ausentes
dados_ausentes = df_peixes.isnull().sum()

print("--- Contagem de Dados Ausentes por Coluna ---")
print(dados_ausentes)
```

Saída Esperada

```
--- Contagem de Dados Ausentes por Coluna ---
local         0
especie       0
quantidade    1
ph_agua       1
dtype: int64
```

O resultado confirma exatamente o que suspeitávamos: as colunas `local` e `especie` estão completas (0 ausentes), enquanto `quantidade` e `ph_agua` possuem um valor faltante. Agora que temos um diagnóstico claro, podemos decidir como tratar essas lacunas.

Estratégia 1: Removendo Dados Ausentes com `.dropna()`

A abordagem mais direta para lidar com dados ausentes é simplesmente removê-los. Se uma linha tem uma informação crucial faltando, talvez seja melhor descartar todo aquele registro para não comprometer a análise. O Pandas torna isso fácil com o método `.dropna()`.

Caso Prático

Para uma análise estatística inicial, decidimos que qualquer registro de coleta que não tenha a contagem de `quantidade` ou o `ph_agua` medido não é útil. Vamos criar um novo DataFrame, `df_completo`, que contém apenas as linhas sem nenhum dado faltante.

Copie e Teste!

```
# Criando uma cópia limpa do DataFrame removendo todas as linhas
# que contenham pelo menos um valor NaN.
df_completo = df_peixes.dropna()

print("--- DataFrame Original ---")
print(df_peixes)

print("\n--- DataFrame Após .dropna() ---")
print(df_completo)
```

Saída Esperada

```
--- DataFrame Original ---
                 local   especie  quantidade  ph_agua
0  Encontro das Águas  Tambaqui        45.0      6.8
1     Frente a Manaus  Tucunaré        22.0      6.5
2       Lago Janauari  Pirarucu        10.0      7.1
3           Parintins  Tambaqui        56.0      NaN
4     Frente a Manaus   Jaraqui        89.0      6.6
5       Lago Janauari  Tucunaré        15.0      7.0
6  Encontro das Águas  Pirarucu         NaN      6.9

--- DataFrame Após .dropna() ---
                 local   especie  quantidade  ph_agua
0  Encontro das Águas  Tambaqui        45.0      6.8
1     Frente a Manaus  Tucunaré        22.0      6.5
2       Lago Janauari  Pirarucu        10.0      7.1
4     Frente a Manaus   Jaraqui        89.0      6.6
5       Lago Janauari  Tucunaré        15.0      7.0
```

Podemos observar que as linhas de índice `3` e `6`, que continham valores do tipo `NaN`, foram completamente removidas.

Fique Alerta!

Remover dados é uma decisão impactante e deve ser feita com cautela. Embora o método .dropna() seja poderoso, seu uso indiscriminado em um conjunto de dados com muitas lacunas espalhadas pode eliminar uma parte significativa de suas informações, empobrecendo a análise. Portanto, avalie sempre o impacto da exclusão antes de descartar registros.

Estratégia 2: Preenchendo Dados Ausentes com `.fillna()`

Em vez de remover informações, muitas vezes a melhor abordagem é preencher as lacunas. Essa técnica é chamada de **imputação**. O método `.fillna()` do Pandas nos oferece várias estratégias inteligentes para isso.

a) Preenchendo com um Valor Fixo A forma mais simples é substituir todos os `NaN`s de uma coluna por um valor constante. Isso é útil quando temos um bom motivo para assumir um valor padrão.

Caso Prático

Decidimos que, para a coluna `quantidade`, um registro faltante provavelmente significa que nenhum peixe daquela espécie foi avistado. Portanto, vamos preencher o `NaN` da coluna `quantidade` com o valor `0`.

Copie e Teste!

```
# Criando uma cópia para não alterar o DataFrame original
df_preenchido = df_peixes.copy()

# Preenchendo o NaN na coluna 'quantidade' com o valor 0
df_preenchido['quantidade'] = df_preenchido['quantidade'].fillna
    (0)

print(df_preenchido)
```

Saída Esperada

```
                local   especie  quantidade  ph_agua
0  Encontro das Águas  Tambaqui        45.0      6.8
1     Frente a Manaus  Tucunaré        22.0      6.5
2       Lago Janauari  Pirarucu        10.0      7.1
3           Parintins  Tambaqui        56.0      NaN
4     Frente a Manaus   Jaraqui        89.0      6.6
5       Lago Janauari  Tucunaré        15.0      7.0
6  Encontro das Águas  Pirarucu         0.0      6.9
```

Observe que apenas o `NaN` da coluna `quantidade` (índice 6) foi substituído por `0.0`. O `NaN` da coluna `ph_agua` permanece.

b) Preenchendo com a Média ou Mediana Para variáveis numéricas como `ph_agua`, preencher com `0` não faria sentido. Uma estratégia estatisticamente mais sólida é preencher o valor faltante com a **média** ou a **mediana** dos valores existentes na coluna. Isso tende a preservar a distribuição estatística dos dados.

Caso Prático

A falha na medição do pH em Parintins foi pontual. Acreditamos que o pH nesse local deveria ser próximo à média dos outros pontos medidos. Vamos calcular a média da coluna `ph_agua` e usar esse valor para preencher a lacuna.

Copie e Teste!

```
# Primeiro, calculamos a média da coluna (o Pandas ignora os NaNs
    no cálculo)
media_ph = df_peixes['ph_agua'].mean()
print(f"A média do pH calculada é: {media_ph:.2f}\n")

# Agora, preenchemos o NaN da coluna 'ph_agua' com a média
df_preenchido['ph_agua'] = df_preenchido['ph_agua'].fillna(
    media_ph)

print(df_preenchido)
```

Saída Esperada

```
A média do pH calculada é: 6.82

                 local   especie  quantidade   ph_agua
0  Encontro das Águas  Tambaqui        45.0  6.800000
1     Frente a Manaus  Tucunaré        22.0  6.500000
2       Lago Janauari  Pirarucu        10.0  7.100000
3           Parintins  Tambaqui        56.0  6.816667
4     Frente a Manaus   Jaraqui        89.0  6.600000
5       Lago Janauari  Tucunaré        15.0  7.000000
6  Encontro das Águas  Pirarucu         0.0  6.900000
```

A linha de índice `3`, que antes tinha `NaN` na coluna `ph_agua`, agora contém o valor da média (`6.816667`), tornando nosso conjunto de dados completo e pronto para análises mais avançadas.

3.2.4 Modificando o DataFrame

Até agora, você aprendeu a carregar seus dados e a inspecioná-los como um detetive que examina uma caixa de evidências. Você sabe o que tem em mãos, qual o tamanho do seu conjunto de dados e onde estão as primeiras ”pistas”, como os dados ausentes. O próximo passo da investigação é organizar e enriquecer essas evidências.

Raramente os dados vêm no formato perfeito para a nossa análise. Muitas vezes, precisamos criar novas informações a partir das existentes (como calcular a densidade de árvores a partir da contagem e da área), corrigir nomes de colunas confusos ou simplesmente limpar a ”mesa de trabalho”, removendo dados irrelevantes. Esta seção é sobre isso: aprender a ”esculpir”seu DataFrame. Vamos transformá-lo de um bloco de dados brutos em uma ferramenta de análise precisa e ajustada às suas perguntas.

Criando Novas Colunas a Partir de Operações

Uma das tarefas mais poderosas e comuns na análise de dados é a criação de novas colunas baseadas em informações que já existem. Isso nos permite derivar novos *insights* e variáveis que não estavam presentes nos dados originais. Felizmente, o Pandas torna essa operação tão intuitiva quanto uma simples atribuição de variável.

Caso Prático

Uma equipe de monitoramento florestal coletou dados sobre diferentes parcelas de uma área de reflorestamento na Amazônia. O arquivo de dados contém o número de árvores e a área de cada parcela em hectares. Para avaliar o sucesso do reflorestamento, a equipe precisa calcular a **densidade de árvores** (árvores por hectare) para cada parcela, uma informação que não está no conjunto de dados original.

Copie e Teste!

```
import pandas as pd

# Criando um DataFrame com os dados da pesquisa de campo
dados_floresta = {
    'id': ['A1', 'A2', 'B1', 'B2'],
    'quantidade': [450, 510, 480, 530],
    'area': [2.0, 2.2, 2.1, 2.3]
}
df_floresta = pd.DataFrame(dados_floresta)

print("--- DataFrame Original ---")
print(df_floresta)

# Criando a nova coluna 'densidade_arvores'
# A operação é feita para todas as linhas de uma só vez!
df_floresta['densidade'] = df_floresta['quantidade'] / df_floresta
   ['area']

print("\n--- DataFrame com a Nova Coluna de Densidade ---")
print(df_floresta)
```

Saída Esperada

```
--- DataFrame Original ---
  id  quantidade  area
0 A1         450   2.0
1 A2         510   2.2
2 B1         480   2.1
3 B2         530   2.3

--- DataFrame com a Nova Coluna de Densidade ---
  id  quantidade  area  densidade
```

```
0 A1          450   2.0 225.000000
1 A2          510   2.2 231.818182
2 B1          480   2.1 228.571429
3 B2          530   2.3 230.434783
```

Observe como a nova coluna `densidade_arvores` foi adicionada ao final do DataFrame. O Pandas aplicou a divisão linha por linha automaticamente, uma funcionalidade conhecida como **vetorização**, que é extremamente eficiente e uma das grandes vantagens de usar bibliotecas como o Pandas em vez de laços `for` manuais.

Renomeando Colunas para Maior Clareza

Os dados que recebemos nem sempre vêm com nomes de colunas claros. Eles podem ser abreviados (como `qtd`), em outro idioma ou conter caracteres que dificultam o acesso. Renomear colunas é um passo fundamental para tornar seu `DataFrame` e seu código mais legíveis e fáceis de manter. Para isso, usamos o método `.rename()`.

Caso Prático

Um sensor automático de qualidade da água gera um arquivo com os nomes das colunas em inglês e de forma abreviada: `'location'`, `'ph_val'`, e `'turb_ntu'`. Para preparar os dados para um relatório em português, precisamos traduzir e tornar esses nomes mais descritivos.

Copie e Teste!

```
import pandas as pd

# DataFrame com nomes de colunas pouco claros
dados_sensor = {
    'location': ['Encontro das Águas', 'Lago Janauari'],
    'ph_val': [6.8, 7.1],
    'turb_ntu': [5.2, 3.1]
}
df_sensor = pd.DataFrame(dados_sensor)

# Usamos .rename() com um dicionário para mapear os nomes antigos
   para os novos
df_sensor_renomeado = df_sensor.rename(columns={
    'location': 'local_coleta',
    'ph_val': 'valor_ph',
    'turb_ntu': 'turbidez_ntu'
})

print("--- DataFrame Original ---")
print(df_sensor)

print("\n--- DataFrame com Colunas Renomeadas ---")
print(df_sensor_renomeado)
```

Saída Esperada

```
--- DataFrame Original ---
             location  ph_val  turb_ntu
0  Encontro das Águas     6.8       5.2
1       Lago Janauari     7.1       3.1

--- DataFrame com Colunas Renomeadas ---
           local_coleta  valor_ph  turbidez_ntu
0  Encontro das Águas       6.8           5.2
1       Lago Janauari       7.1           3.1
```

Fique Alerta!

Por padrão, o método `.rename()` não modifica o DataFrame original; ele retorna uma **cópia** com as alterações. Por isso, salvamos o resultado em uma nova variável (`df_sensor_renomeado`). Uma maneira de modificar o DataFrame original diretamente é configurar a função com o parâmetro `inplace=True`. No entanto, trabalhar com cópias costuma ser uma prática mais segura, especialmente para iniciantes.

Removendo Colunas Desnecessárias

Muitas vezes, os conjuntos de dados contêm colunas que não são relevantes para a nossa análise específica. Remover essas colunas simplifica o `DataFrame`, economiza memória e nos ajuda a focar no que realmente importa. A maneira mais comum e segura de fazer isso é com o método `.drop()`.

Caso Prático

Nosso DataFrame de monitoramento de peixes contém um código interno de coleta (`'cod_coleta'`) e a data no formato timestamp Unix (`'timestamp'`), que não serão usados na nossa análise de biodiversidade. Vamos remover essas colunas para simplificar nossa tabela.

Copie e Teste!

```python
import pandas as pd

# DataFrame com colunas a serem removidas
dados_completos = {
    'cod_coleta': ['C01', 'C02', 'C03'],
    'local': ['Parintins', 'Maués', 'Itacoatiara'],
    'especie': ['Tambaqui', 'Pirarucu', 'Tucunaré'],
    'timestamp': [1672531200, 1672617600, 1672704000]
}
df_completo = pd.DataFrame(dados_completos)

# Removendo uma lista de colunas com o método .drop()
```

```
# O argumento 'axis=1' informa ao Pandas que queremos remover
   colunas, não linhas
df_simplificado = df_completo.drop(columns=['cod_coleta', '
   timestamp'])

print("--- DataFrame Original ---")
print(df_completo)

print("\n--- DataFrame Após Remover Colunas ---")
print(df_simplificado)
```

Saída Esperada

```
--- DataFrame Original ---
  cod_coleta         local   especie   timestamp
0        C01     Parintins  Tambaqui  1672531200
1        C02         Maués  Pirarucu  1672617600
2        C03   Itacoatiara  Tucunaré  1672704000

--- DataFrame Após Remover Colunas ---
           local   especie
0      Parintins  Tambaqui
1          Maués  Pirarucu
2    Itacoatiara  Tucunaré
```

O método `.drop()` é a ferramenta padrão para remoção, pois é explícito e flexível. Dominar a criação, renomeação e remoção de colunas, é essencial para o processo de *data munging* ou preparação de dados, que constitui a maior parte do trabalho de um analista (McKinney, 2018).

3.3 Análise e Agregação de Dados

Nos tópicos anteriores, você aprendeu a carregar, inspecionar e limpar seus dados. Foi como um explorador que, após uma longa expedição, organiza todas as amostras e anotações em uma mesa de laboratório. Agora que tudo está em ordem, começa a fase mais emocionante e desafiadora da descoberta, a análise das informações. Ter dados limpos e organizados é essencial, mas o verdadeiro valor está nas histórias que eles podem contar e nas perguntas que podem responder.

Esta seção é dedicada a transformar seu `DataFrame` de uma simples tabela de informações em uma fonte rica de conhecimento. Pense nesta etapa como o momento em que o explorador começa a conectar os pontos: calcular a média de altitude onde uma espécie de planta foi encontrada ou agrupar todas as amostras de uma mesma região para entender suas características comuns. Para realizar essa tarefa, vamos focar em duas das técnicas mais poderosas e fundamentais do Pandas:

- **Cálculos Estatísticos Básicos:** Aprenderemos a extrair rapidamente as principais métricas de nossos dados. Utilizando funções simples como `.mean()`, `.median()` e

`.sum()`, você poderá responder a perguntas como ”Qual foi a temperatura média registrada?”ou ”Qual a quantidade total de peixes coletados?”. Esses cálculos são o primeiro passo para entender as tendências centrais, dispersões e a magnitude dos seus dados.

- **Agrupamento de Dados (groupby):** Esta é, talvez, uma das funcionalidades mais poderosas do Pandas. A técnica de `groupby` nos permite ”dividir”nosso conjunto de dados em grupos com base em uma categoria e, em seguida, aplicar uma análise a cada grupo separadamente. É a ferramenta perfeita para responder a perguntas comparativas, como ”Qual local teve a maior média de turbidez na água?”ou ”Qual a produção total de açaí por estado?”.

Dominar essas técnicas de agregação é o que permite ir além da simples observação dos dados e começar a extrair conclusões significativas, revelando padrões e tendências que estavam escondidos sob a superfície (McKinney, 2018). Vamos começar a transformar nossos números em verdadeiros *insights*.

3.3.1 Cálculos Estatísticos Básicos: O Primeiro Raio-X dos Seus Dados

Depois de limpar um conjunto de dados, o primeiro passo de um analista é obter um resumo numérico. A estatística descritiva nos ajuda a responder perguntas fundamentais de forma rápida. O Pandas transforma essa tarefa, que seria complexa, em operações de uma única linha. Vamos usar nosso `DataFrame` de monitoramento de peixes, já limpo, para explorar essas ferramentas.

Caso Prático

Vamos continuar nossa análise com os dados de peixes, agora focando nas colunas numéricas `quantidade` e `ph_agua` para extrair nossas primeiras estatísticas.

Copie e Teste!

```
import pandas as pd
import numpy as np

# Recriando o DataFrame já com os dados ausentes tratados
dados = {
    'local': ['Encontro das Águas', 'Frente a Manaus', 'Lago
   Janauari', 'Parintins', 'Frente a Manaus', 'Lago Janauari', '
   Encontro das Águas'],
    'especie': ['Tambaqui', 'Tucunaré', 'Pirarucu', 'Tambaqui', '
   Jaraqui', 'Tucunaré', 'Pirarucu'],
    'quantidade': [45.0, 22.0, 10.0, 56.0, 89.0, 15.0, 0.0], # NaN
    preenchido com 0
    'ph_agua': [6.8, 6.5, 7.1, 6.82, 6.6, 7.0, 6.9] # NaN
   preenchido com a média
}
df_peixes = pd.DataFrame(dados)

# 1. Qual a quantidade total de peixes avistados? .sum()
total_peixes = df_peixes['quantidade'].sum()
```

```
print(f"Total de peixes avistados: {total_peixes}")

# 2. Qual foi a média de pH da água? .mean()
media_ph = df_peixes['ph_agua'].mean()
print(f"Média do pH da água: {media_ph:.2f}")

# 3. Qual o valor mediano da contagem de peixes? .median()
mediana_peixes = df_peixes['quantidade'].median()
print(f"Mediana da quantidade de peixes: {mediana_peixes}")

# 4. Qual foi a maior e a menor quantidade de peixes registrada? .
   max() e .min()
max_peixes = df_peixes['quantidade'].max()
min_peixes = df_peixes['quantidade'].min()
print(f"Maior contagem em um local: {max_peixes}")
print(f"Menor contagem em um local: {min_peixes}")

# 5. Quantas medições de pH foram realizadas? .count()
contagem_medicoes = df_peixes['ph_agua'].count()
print(f"Total de medições de pH: {contagem_medicoes}")
```

Saída Esperada

```
Total de peixes avistados: 237.0
Média do pH da água: 6.82
Mediana da quantidade de peixes: 22.0
Maior contagem em um local: 89.0
Menor contagem em um local: 0.0
Total de medições de pH: 7
```

Conhecendo um pouco mais!

Sugestão de Imagem 1: Inserir aqui um diagrama simples e intuitivo. Deveria mostrar uma série de números e indicar visualmente o que são a média (o ponto de equilíbrio), a mediana (o valor do meio), o mínimo e o máximo.

Fique Alerta!

Média vs. Mediana
A **média** (`.mean()`) é a soma de todos os valores dividida pela quantidade de valores. Ela é muito útil, mas pode ser enganosa se houver valores extremos (*outliers*). Por exemplo, na lista `[10, 20, 30, 40, 1000]`, a média é 220, um valor que não representa bem a maioria dos dados.
A **mediana** (`.median()`) é o valor que está exatamente no meio da lista de dados, após ela ser ordenada. Na mesma lista, a mediana é 30, que é um valor muito mais representativo da ”tendência central”dos dados. Escolher entre média e mediana é uma decisão importante na análise de dados.

A Ferramenta Definitiva: Resumo Estatístico com `.describe()`

Calcular cada métrica individualmente é útil, mas o que aconteceria se você quisesse um resumo rápido de todas as colunas numéricas de uma só vez? O Pandas nos oferece uma função incrivelmente poderosa para isso. Com um único comando, o método `.describe()` calcula as principais estatísticas descritivas para todas as colunas numéricas do seu DataFrame.

- **count:** Contagem de valores não nulos.
- **mean:** A média.
- **std:** O desvio padrão.
- **min:** O valor mínimo.
- **25%:** O primeiro quartil.
- **50%:** A mediana (o segundo quartil).
- **75%:** O terceiro quartil.
- **max:** O valor máximo.

Caso Prático

Vamos aplicar o `.describe()` ao nosso DataFrame `df_peixes` para obter um panorama completo das nossas medições de campo com um único comando.

Copie e Teste!

```
# Gerando o resumo estatístico completo
resumo_estatistico = df_peixes.describe()

print("--- Resumo Estatístico do DataFrame ---")
print(resumo_estatistico)
```

Saída Esperada

```
--- Resumo Estatístico do DataFrame ---
       quantidade   ph_agua
count    7.000000  7.000000
mean    33.857143  6.817143
std     31.144823  0.222389
min      0.000000  6.500000
25%     12.500000  6.700000
50%     22.000000  6.820000
75%     50.500000  6.950000
max     89.000000  7.100000
```

Em segundos, temos uma visão completa. Podemos ver, por exemplo, que a contagem de peixes (`quantidade`) varia muito (o desvio padrão, `std`, é alto, quase igual à média), enquanto o pH da água (`ph_agua`) é bem mais estável (desvio padrão baixo).

Essas ferramentas de cálculo estatístico são a base da análise exploratória de dados. Elas nos permitem passar da simples observação para a quantificação, o primeiro passo para descobrir os padrões escondidos em nossos dados. Agora que você sabe como resumir seus dados, o próximo passo é aprender a dividi-los em grupos para fazer comparações ainda mais ricas e detalhadas.

3.3.2 Agrupamento de Dados (groupby)

Nos tópicos anteriores, você aprendeu a carregar, inspecionar e limpar seus dados. Foi como um explorador que, após uma longa expedição, organiza todas as suas amostras e anotações em uma mesa de laboratório. Agora que tudo está em ordem, começa a fase mais emocionante da descoberta, a análise das informações para encontrar padrões.

Ter dados limpos e organizados é essencial, mas o verdadeiro valor está nas histórias que eles podem contar e nas perguntas que podem responder. Até agora, nossos cálculos como `.mean()` ou `.sum()` foram aplicados ao conjunto de dados como um todo. Mas, na maioria das vezes, nossas perguntas são mais específicas e comparativas:

- "Qual a produção **média** de açaí **por estado**?"
- "Qual a quantidade **total** de peixes avistados **para cada espécie**?"
- "Qual local de coleta apresentou a **maior média** de pH na água?"

Para responder a esse tipo de pergunta, precisamos de uma ferramenta que nos permita dividir nossos dados em grupos e analisar cada um deles separadamente. Pense nesta etapa como o momento em que o explorador começa a conectar os pontos, agrupar todas as amostras de uma mesma região para entender suas características comuns.

A Estratégia "Dividir-Aplicar-Combinar"(Split-Apply-Combine)

O poder do `.groupby()` pode ser entendido por uma estratégia de três passos conhecida como "Dividir-Aplicar-Combinar"(*Split-Apply-Combine*). Imagine que você tem uma cesta cheia de frutas amazônicas misturadas: açaí, cupuaçu e tucumã. Sua tarefa é descobrir o peso médio de cada tipo de fruta. Como você faria isso?

1. **Dividir (Split):** Você primeiro separa as frutas em três cestas menores, uma para cada tipo: uma só com açaí, outra só com cupuaçu e a terceira só com tucumã.
2. **Aplicar (Apply):** Em seguida, você pega cada cesta individualmente e aplica uma operação: você pesa todas as frutas daquela cesta e calcula o peso médio. Você faz isso para a cesta de açaí, depois para a de cupuaçu e, por fim, para a de tucumã.
3. **Combinar (Combine):** Finalmente, você pega os resultados de cada cesta (o peso médio de cada fruta) e os combina em uma única tabela-resumo, mostrando o resultado final da sua análise.

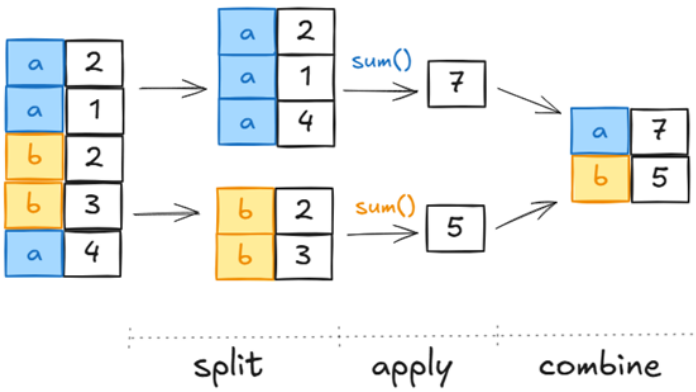

Figura 3.4: Diagrama de representação da estratégia Dividir-Aplicar-Combinar
Fonte: Michele Lozada, 2024

Agrupando por uma Única Coluna

A forma mais comum de agrupamento é baseada nos valores de uma única coluna categórica. Vamos voltar ao nosso exemplo de monitoramento de peixes para ver isso na prática.

Caso Prático

Utilizando nosso DataFrame `df_peixes`, já limpo e com os dados ausentes tratados, queremos descobrir a **quantidade total** de indivíduos contados para **cada espécie**. Isso nos ajudará a entender quais espécies foram mais abundantes em nossas coletas.

Copie e Teste!

```
import pandas as pd
import numpy as np

# Recriando o DataFrame já com os dados ausentes tratados
dados = {
    'local': ['Encontro das Águas', 'Frente a Manaus', 'Lago
   Janauari', 'Parintins', 'Frente a Manaus', 'Lago Janauari', '
   Encontro das Águas'],
    'especie': ['Tambaqui', 'Tucunaré', 'Pirarucu', 'Tambaqui', '
   Jaraqui', 'Tucunaré', 'Pirarucu'],
    'quantidade': [45.0, 22.0, 10.0, 56.0, 89.0, 15.0, 0.0], # NaN
    preenchido com 0
    'ph_agua': [6.8, 6.5, 7.1, 6.82, 6.6, 7.0, 6.9] # NaN
   preenchido com a média
}
df_peixes = pd.DataFrame(dados)

# 1. Dividir: Agrupamos o DataFrame pela coluna 'especie'
# 2. Aplicar: Selecionamos a coluna 'quantidade' e aplicamos a
   soma (.sum())
# 3. Combinar: O Pandas combina os resultados em uma nova Series
contagem_por_especie = df_peixes.groupby('especie')['quantidade'].
   sum()

print("--- Contagem Total de Peixes por Espécie ---")
print(contagem_por_especie)
```

Saída Esperada

```
--- Contagem Total de Peixes por Espécie ---
especie
Jaraqui       89.0
Pirarucu      10.0
Tambaqui     101.0
Tucunaré      37.0
Name: quantidade, dtype: float64
```

O resultado é uma nova `Series` onde o índice são as espécies (as categorias pelas quais agrupamos) e os valores são o resultado da soma. Agora, podemos ver claramente que o Tambaqui foi a espécie com o maior número total de indivíduos contados em todas as coletas.

Conhecendo um pouco mais!

Quando você executa `df.groupby('coluna')`, o Pandas não retorna imediatamente um DataFrame, mas sim um objeto especial chamado `DataFrameGroupBy`. Pense nele como uma representação intermediária dos seus dados, já divididos em grupos, mas esperando por uma instrução do que fazer a seguir (o passo "Aplicar"). É somente quando você aplica uma função de agregação (como `.sum()`, `.mean()`, `.count()`, etc.) que o cálculo é realizado e o resultado final é combinado e exibido.

Aplicando Múltiplas Agregações com `.agg()`

E se quisermos mais do que apenas a soma? Muitas vezes, para cada grupo, nos interessa calcular várias estatísticas de uma só vez. Para isso, usamos o método **`.agg()`** (de *aggregate*), que nos permite aplicar uma lista de funções de agregação a cada grupo.

Caso Prático

Para cada espécie de peixe, a equipe de pesquisa quer um resumo mais completo. Eles precisam saber não apenas a soma (`sum`), mas também a contagem média de indivíduos por coleta (`mean`) e o maior número de indivíduos encontrados em uma única coleta (`max`).

Copie e Teste!

```
# Agrupamos por 'especie', selecionamos 'quantidade' e aplicamos .
    agg()
resumo_especies = df_peixes.groupby('especie')['quantidade'].agg([
    'sum', 'mean', 'max'])

print("--- Resumo Agregado por Espécie ---")
print(resumo_especies)
```

Saída Esperada

```
--- Resumo Agregado por Espécie ---
            sum   mean   max
especie
Jaraqui    89.0  89.00  89.0
Pirarucu   10.0   5.00  10.0
Tambaqui  101.0  50.50  56.0
Tucunaré   37.0  18.50  22.0
```

Desta vez, o resultado é um `DataFrame`! As linhas ainda são as espécies, mas agora as colunas são as funções de agregação que pedimos. Essa tabela nos dá uma visão muito mais rica. Por exemplo, enquanto o Jaraqui teve o maior número de avistamentos em uma única

coleta (`max` de 89.0), o Tambaqui teve uma média de avistamentos por coleta (`mean` de 50.5) muito superior à do Pirarucu.

Agrupando por Múltiplas Colunas

A análise pode ficar ainda mais detalhada. E se quisermos entender as características não apenas por espécie, mas por **espécie em cada local**? O `.groupby()` aceita uma lista de nomes de colunas, criando grupos baseados na combinação única dos valores dessas colunas.

Caso Prático

Os pesquisadores querem uma análise ainda mais granular. Eles precisam saber a **quantidade média** de peixes avistados para cada **espécie** em cada **local** de coleta. Isso ajudará a identificar se certas espécies são mais comuns em locais específicos.

Copie e Teste!

```
# Agrupamos por uma lista de colunas: ['local', 'especie']
media_por_local_especie = df_peixes.groupby(['local', 'especie'])[
    'quantidade'].mean()

print("--- Média de Peixes por Local e Espécie ---")
print(media_por_local_especie)
```

Saída Esperada

```
--- Média de Peixes por Local e Espécie ---
local                  especie
Encontro das Águas     Pirarucu     0.0
                       Tambaqui    45.0
Frente a Manaus        Jaraqui     89.0
                       Tucunaré    22.0
Lago Janauari          Pirarucu    10.0
                       Tucunaré    15.0
Parintins              Tambaqui    56.0
Name: quantidade, dtype: float64
```

O resultado agora tem um **`MultiIndex`** (índice hierárquico). Isso significa que os grupos são formados pela combinação das duas colunas. Com essa tabela, podemos facilmente comparar, por exemplo, que a média de Tucunarés em ”Frente a Manaus”(22.0) foi maior do que em ”Lago Janauari”(15.0).

Fique Alerta!

O conceito de agrupamento é uma das transições mais importantes da manipulação para a análise de dados. É a ferramenta que permite que você pare de olhar para os dados como uma lista de registros individuais e comece a vê-los como grupos com características e comportamentos coletivos.

Dominar a agregação de dados é o que permite extrair conclusões significativas, revelando padrões e tendências que estavam escondidos. Agora que você sabe como calcular e agrupar seus dados para criar resumos poderosos, o próximo passo natural é aprender a **visualizar** esses resultados. Afinal, uma imagem, ou um gráfico, muitas vezes conta uma história de forma muito mais clara e impactante do que uma tabela de números.

3.4 Visualização de Dados com Matplotlib

Nos tópicos anteriores, você se tornou um verdadeiro explorador de dados. Aprendeu a usar o Pandas para carregar suas anotações de campo, limpar a bagunça, organizar tudo em tabelas e até mesmo calcular estatísticas e resumos que revelaram os primeiros segredos escondidos nos números. Sua mesa de trabalho agora está cheia de informações valiosas: a média de pH da água, a quantidade total de peixes avistados por espécie, a produção de açaí em cada local.

Essas tabelas e resumos são a base de qualquer grande descoberta. Contudo, se você apresentar uma tabela cheia de números para a sua equipe, eles podem levar um tempo para entender a história que você encontrou. E se você pudesse mostrar essa história em um piscar de olhos?

É aqui que deixamos de ser apenas exploradores para nos tornarmos contadores de histórias. A visualização de dados é a arte e a ciência de transformar análises numéricas em representações gráficas que nosso cérebro consegue interpretar de forma instantânea e intuitiva. Um bom gráfico não apenas apresenta resultados; ele revela padrões, destaca tendências e comunica *insights* de uma maneira que nenhuma tabela consegue igualar.

Nesta seção, vamos mergulhar na biblioteca mais fundamental e amplamente utilizada para a criação de gráficos em Python: a **Matplotlib**. Se o Pandas foi o seu canivete suíço para manipular os dados, a Matplotlib será o seu estúdio de arte para dar vida a eles. Com ela, você aprenderá a transformar suas análises em ferramentas de comunicação poderosas e profissionais. Nossa jornada pela visualização de dados será dividida em três partes essenciais:

- **Gráficos Essenciais: Linhas e Barras:** Começaremos com os dois tipos de gráficos mais importantes e versáteis do arsenal de qualquer analista. Aprenderemos a criar gráficos de linhas, perfeitos para mostrar a evolução de uma variável ao longo do tempo, e gráficos de barras, ideais para comparar quantidades entre diferentes categorias.

- **Customizando suas Visualizações:** Em seguida, aprenderemos a refinar nossos gráficos para torná-los mais claros, profissionais e informativos. Veremos como adicionar títulos, rotular os eixos, incluir legendas e ajustar cores, transformando um gráfico básico em uma visualização pronta para um relatório ou apresentação.

- **Gráficos Estatísticos:** Para aprofundar nossa análise, exploraremos gráficos que nos ajudam a entender a *distribuição* dos nossos dados e a *relação* entre diferentes variáveis. Vamos mergulhar em ferramentas poderosas como o **histograma**, para ver a frequência dos dados, o **boxplot**, para identificar a dispersão e os *outliers*, e o **diagrama de dispersão**, essencial para investigar se duas variáveis se movem juntas.

A visualização de dados é uma ferramenta essencial de análise que nos permite identificar tendências, anomalias e padrões que seriam quase impossíveis de perceber apenas olhando para os números (McKinney, 2018). Prepare-se para transformar suas descobertas em histórias visuais claras, impactantes e convincentes.

3.4.1 Gráficos Essenciais: Linhas e Barras

Começaremos com os dois tipos de gráficos mais importantes e versáteis do arsenal de qualquer analista: os gráficos de linhas e os de barras. Aprenderemos a criá-los de forma integrada com o Pandas, transformando nossas tabelas de dados em visualizações claras com poucas linhas de código.

Conhecendo um pouco mais!

Para usar a Matplotlib, primeiro você precisa instalá-la no seu ambiente Python. Se você estiver usando uma distribuição como o Anaconda, ela já vem incluída. Caso contrário, abra seu terminal ou prompt de comando e digite:

```
pip install matplotlib
```

A Anatomia de um Gráfico

Antes de criarmos nosso primeiro gráfico, é útil conhecer suas partes principais. Assim como um mapa tem uma legenda e uma escala, um gráfico tem componentes que nos ajudam a entendê-lo.

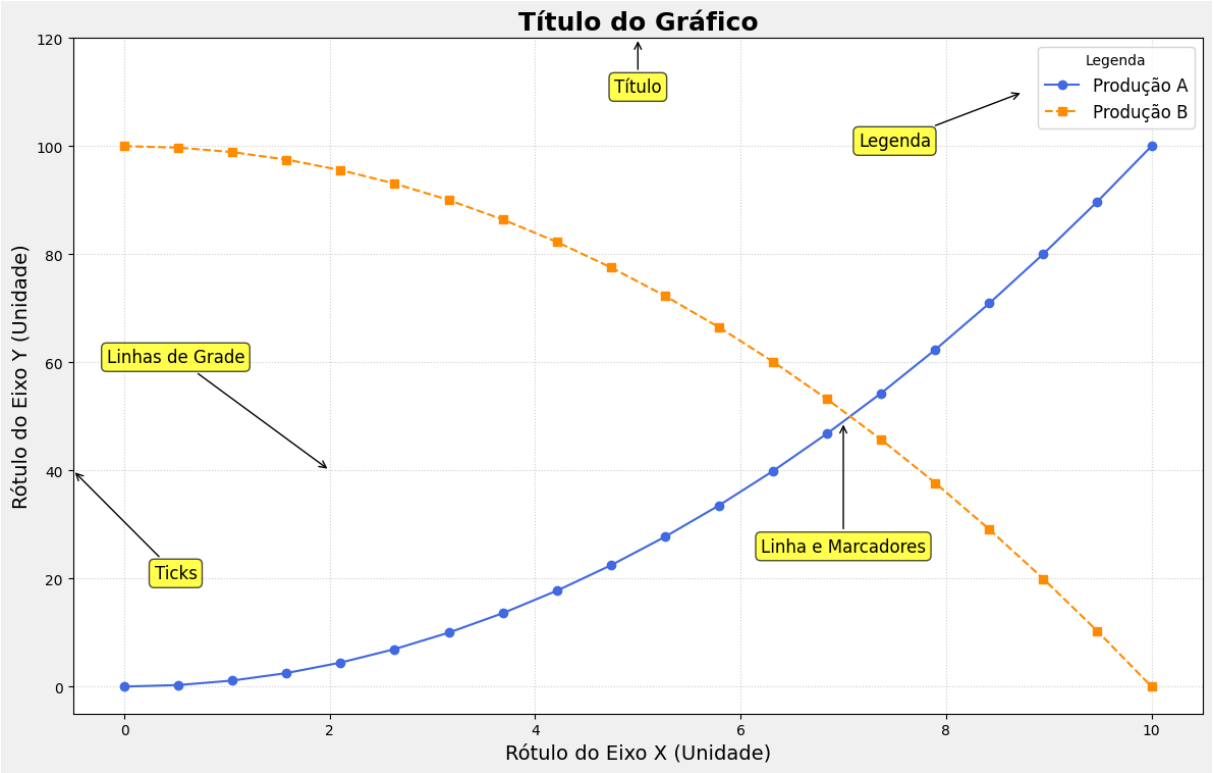

Figura 3.5: Anatomia de um gráfico

Gráficos de Linha: Mostrando Tendências ao Longo do Tempo

Um gráfico de linhas é a melhor ferramenta para visualizar como uma variável numérica muda ao longo de um intervalo contínuo, como o tempo. Ele conecta uma série de pontos de dados com uma linha, tornando muito fácil enxergar tendências, picos e vales.

Caso Prático

Uma estação de monitoramento na Bacia Amazônica registrou o nível médio de um rio a cada mês durante um ano. Os dados foram salvos em um arquivo `nivel_rio.csv`. O objetivo é visualizar a variação do nível do rio ao longo do ano para identificar os períodos de cheia e vazante.

Saída Esperada

```
mes,nivel_m
Janeiro,18.5
Fevereiro,22.1
Marco,25.7
Abril,28.2
Maio,29.5
Junho,29.1
Julho,27.3
Agosto,24.6
Setembro,21.0
Outubro,18.9
Novembro,17.8
Dezembro,18.1
```

Copie e Teste!

```
import pandas as pd
import matplotlib.pyplot as plt

# Carregar os dados para um DataFrame
df_rio = pd.read_csv('nivel_rio.csv')

# Criar o gráfico de linha usando o método .plot() do Pandas
# O Pandas usa o Matplotlib por baixo dos panos para criar o
   gráfico
df_rio.plot(kind='line', x='mes', y='nivel_m', marker='o')

# Customizando o gráfico com funções do Matplotlib
plt.title('Nível Médio Mensal do Rio na Bacia Amazônica')
plt.xlabel('Mês')
plt.ylabel('Nível (metros)')
plt.xticks(ticks=df_rio.index, labels=df_rio['mes'], rotation=45)
   # Incluí os meses como rótulo do eixo x e rotaciona para melhor
    vizualização
plt.grid(True) # Adiciona uma grade para facilitar a leitura
plt.tight_layout() # Ajusta o gráfico para caber tudo

# Exibir o gráfico
plt.show()
```

Ao executar o código acima, uma janela aparecerá com o gráfico de linha. Observe como o método `.plot()` do DataFrame simplifica a criação do gráfico. A integração entre Pandas e Matplotlib nos permite passar de dados a uma visualização com pouquíssimo esforço.

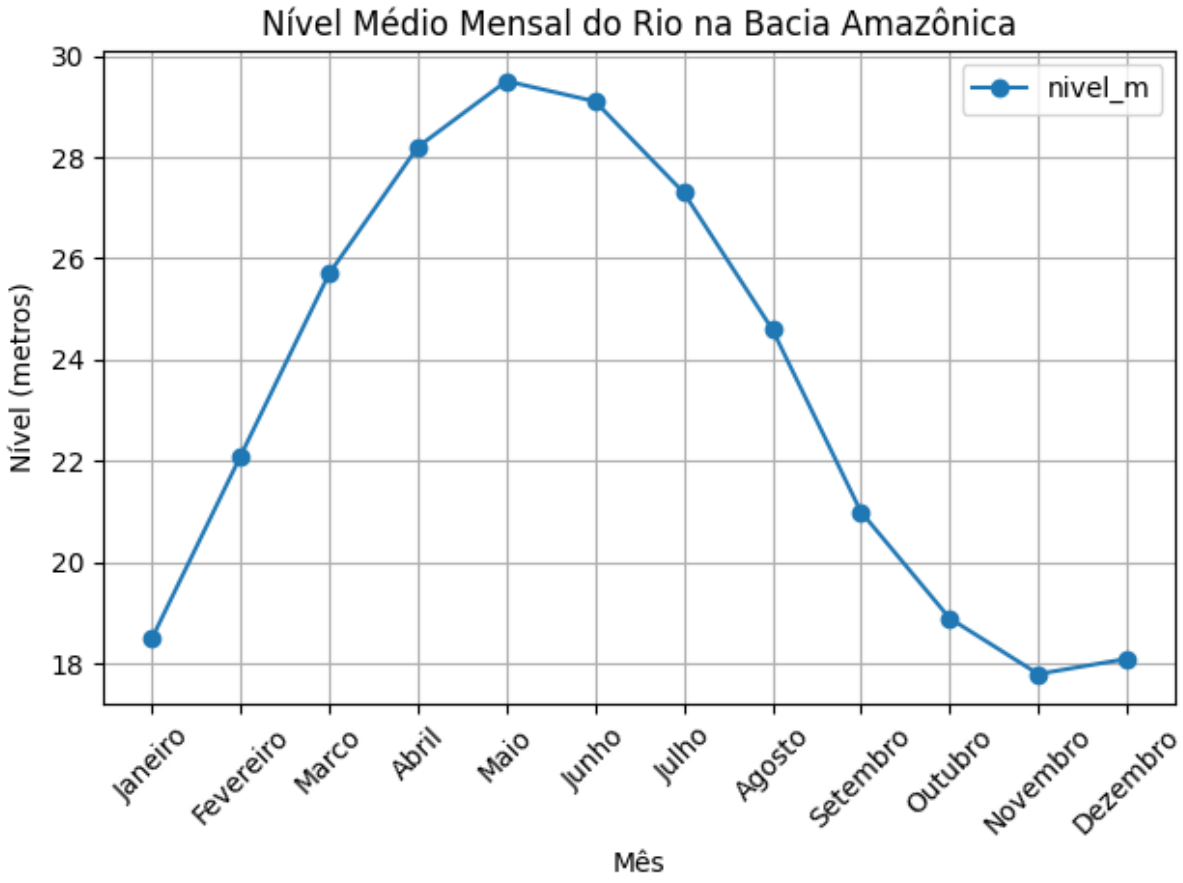

Figura 3.6: Gráfico de linhas gerado pelo Matplotlib

A figura 3.6 nos conta uma história clara e imediata que seria difícil de perceber apenas olhando os números na tabela. Podemos observar visualmente o padrão sazonal do rio:

- **Período de Cheia:** O nível da água sobe de forma constante a partir de Janeiro, atingindo seu ponto máximo por volta de Maio e Junho.
- **Período de Vazante:** A partir de Julho, a tendência se inverte, e o nível do rio começa a baixar, chegando aos seus menores valores no final do ano.

Essa visualização representa perfeitamente o conhecido ciclo de **cheia** (enchente) e **vazante** (seca) dos rios na Bacia Amazônica. A capacidade de transformar uma tabela de dados em um insight visual e de fácil compreensão é um dos principais objetivos da análise de dados.

A clareza desta visualização vem da integração entre o Pandas e o Matplotlib. Enquanto `df_rio.plot()` cria a base do gráfico diretamente a partir dos dados, as funções da biblioteca `matplotlib.pyplot` (que importamos como `plt`) nos permitem refinar e anotar o resultado, transformando um gráfico simples em uma ferramenta de comunicação profissional e eficaz.

Gráficos de Barras: Comparando Categorias

Enquanto os gráficos de linha são ótimos para o tempo, os **gráficos de barras** são perfeitos para comparar quantidade entre diferentes grupos ou categorias. Essa é a ferramenta ideal para visualizar os resultados dos agrupamentos através da função `.groupby()` que aprendemos no tópico anterior.

Caso Prático

Usando nosso DataFrame de coleta de peixes (`df_peixes`), queremos comparar a quantidade total de indivíduos contados para cada espécie. Isso nos ajudará a entender visualmente qual espécie foi a mais abundante em todas as coletas.

Copie e Teste!

```
import pandas as pd
import matplotlib.pyplot as plt

# Recriando o DataFrame de peixes já limpo
dados = {
    'local': ['Encontro das Águas', 'Frente a Manaus', 'Lago
   Janauari', 'Parintins', 'Frente a Manaus', 'Lago Janauari', '
   Encontro das Águas'],
    'especie': ['Tambaqui', 'Tucunaré', 'Pirarucu', 'Tambaqui', '
   Jaraqui', 'Tucunaré', 'Pirarucu'],
    'quantidade': [45.0, 22.0, 10.0, 56.0, 89.0, 15.0, 0.0]
}
df_peixes = pd.DataFrame(dados)

# 1. Agrupar os dados para obter a soma por espécie (Análise)
contagem_por_especie = df_peixes.groupby('especie')['quantidade'].
   sum()

# 2. Criar o gráfico de barras a partir do resultado (Visualização
   )
contagem_por_especie.plot(kind='bar', color=['skyblue', 'salmon',
   'lightgreen', 'gold'])

# 3. Customizar e exibir o gráfico
plt.title('Contagem Total de Peixes por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Quantidade Total Contada')
plt.xticks(rotation=0) # Mantém os nomes das espécies na
   horizontal
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

A figura 3.7 mostra de forma imediata que ”Tambaqui”e ”Jaraqui”foram as espécies com a maior contagem total, enquanto ”Pirarucu”teve a menor. Essa conclusão é muito mais rápida de se obter a partir do gráfico do que da tabela de números.

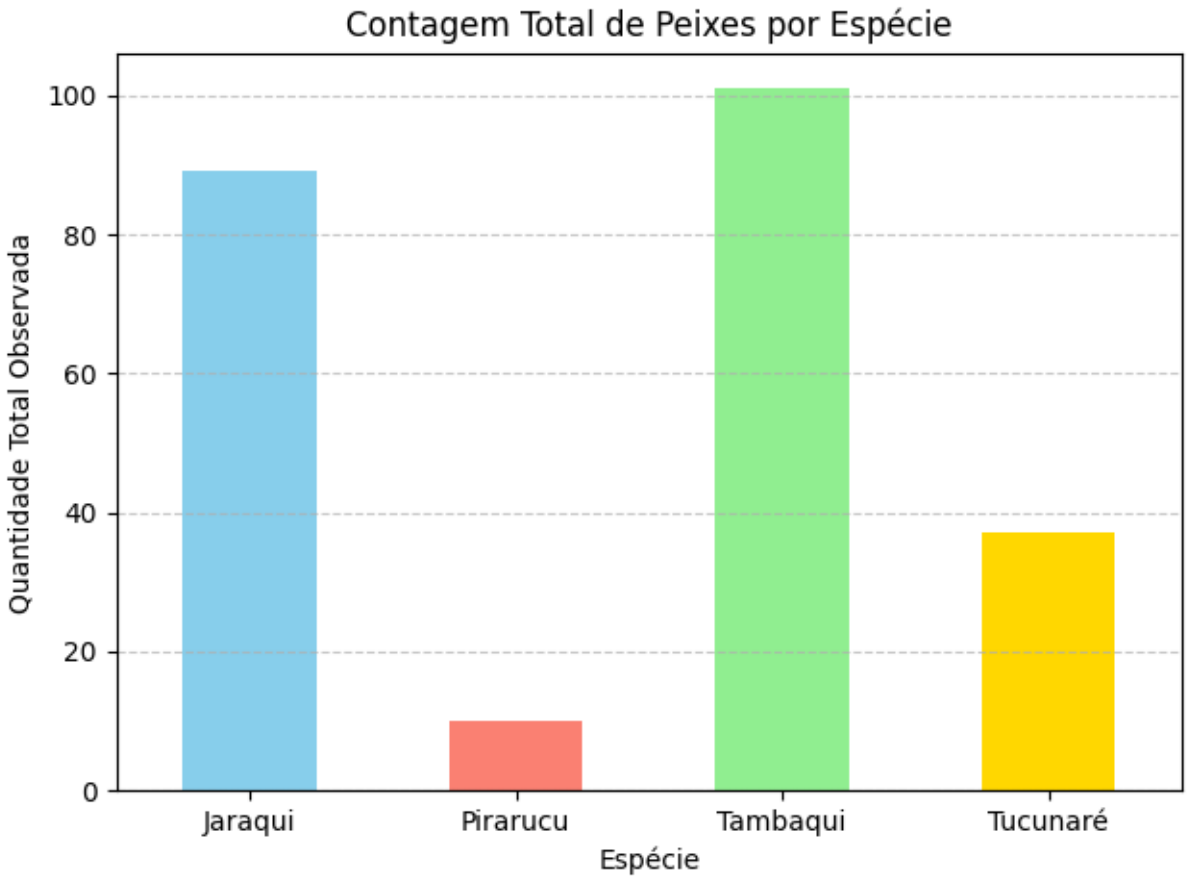

Figura 3.7: Gráfico de barras agrupado por espécie

Fique Alerta!

Quando os nomes das categorias (os rótulos do eixo X) são muito longos, eles podem se sobrepor e dificultar a leitura. Nesses casos, além de rotacionar os rótulos (`plt.xticks(rotation=45)`), uma ótima alternativa é usar um **gráfico de barras horizontais**. Basta trocar `kind='bar'` por `kind='barh'`. Isso dá mais espaço para os nomes das categorias e pode deixar seu gráfico muito mais limpo.

Dominar esses dois tipos de gráficos essenciais já lhe dá o poder de visualizar uma vasta gama de análises de dados. No próximo tópico, vamos aprofundar nossas habilidades, aprendendo a customizar cada detalhe dos nossos gráficos para torná-los ainda mais profissionais e informativos.

3.4.2 Customizando suas Visualizações: Da Análise à Narrativa

Nos tópicos anteriores, aprendemos a criar gráficos básicos. Esse é o primeiro passo. Agora, vamos transformar esses rascunhos em ferramentas de comunicação profissionais. Um gráfico padrão mostra os dados; um gráfico customizado conta uma história, guia o leitor através das suas descobertas e destaca os insights mais importantes.

Nesta seção, vamos aprofundar nossas habilidades com a Matplotlib. Usaremos um exemplo contínuo para adicionar camadas de informação e contexto, e depois exploraremos como comparar dados e escolher a melhor forma visual para a sua análise.

Aprimorando um Gráfico: Um Estudo de Caso do Nível do Rio

Vamos usar nosso exemplo do nível do rio para mostrar um fluxo de trabalho completo, indo de um gráfico simples a uma visualização rica em informações.

Caso Prático

Uma estação de monitoramento registrou o nível médio de um rio na Amazônia a cada mês. Nosso objetivo é transformar esses dados brutos em uma ferramenta visual de monitoramento de risco de enchentes, que seja clara para a comunidade ribeirinha e para a defesa civil, respeitando alguns critérios.

Copie e Teste!

```
import pandas as pd
import matplotlib.pyplot as plt

# --- DADOS ---
dados_rio = {
    'mes': ['Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun',
            'Jul', 'Ago', 'Set', 'Out', 'Nov', 'Dez'],
    'nivel_m': [18.5, 22.1, 25.7, 28.2, 29.5, 29.1, 27.3, 24.6,
   21.0, 18.9, 17.8, 18.1]
}
df_rio = pd.DataFrame(dados_rio)

# --- PASSO 1: Usando um Estilo Profissional ---
# Esta única linha melhora drasticamente a aparência do gráfico.
plt.style.use('ggplot')

# --- PASSO 2: Criação da Figura e do Gráfico Base ---
# `plt.subplots` cria a "tela" (fig) e os "eixos" (ax) separados.
fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(df_rio['mes'], df_rio['nivel_m'], marker='o', linestyle='-
   ', color='royalblue', label='Nível Médio do Rio')

# --- PASSO 3: Adicionando Contexto com Linhas de Referência ---
# `axhline` desenha uma linha para marcar um limiar importante.
ax.axhline(y=28.0, color='red', linestyle='--', linewidth=2, label
   ='Nível de Alerta de Enchente')

# --- PASSO 4: Adicionando Narrativa com Anotações ---
# `annotate` permite adicionar texto e setas para destacar pontos.
pico_mes_index = 4 # Índice do mês de Maio
pico_valor = 29.5
ax.annotate('Pico da Cheia', xy=(pico_mes_index, pico_valor),
   xytext=(pico_mes_index - 2, pico_valor - 0.5), arrowprops=dict(
   facecolor='black', shrink=0.05, width=1, headwidth=8), fontsize
   =12, fontweight='bold')
```

```
# --- PASSO 5: Títulos, Rótulos e Legendas ---
ax.set_title('Monitoramento do Nível do Rio na Bacia Amazônica
    (2024)', fontsize=16)
ax.set_ylabel('Nível (metros)', fontsize=12)
ax.set_xlabel('Mês', fontsize=12)
plt.xticks(rotation=45)

# Ativa a legenda para identificar as linhas plotadas com 'label'
ax.legend()

# Ajustes finais para garantir que nada seja cortado
plt.tight_layout()
plt.show()
```

Ao seguir esses passos, transformamos um gráfico de linha básico em uma ferramenta de análise completa. Ele não apenas mostra os dados, mas também:

- **Tem aparência profissional** (graças ao `plt.style.use`).
- **Fornece contexto crucial** (a linha de alerta de enchente).
- **Conta uma história**, destacando o evento mais importante (o pico da cheia).

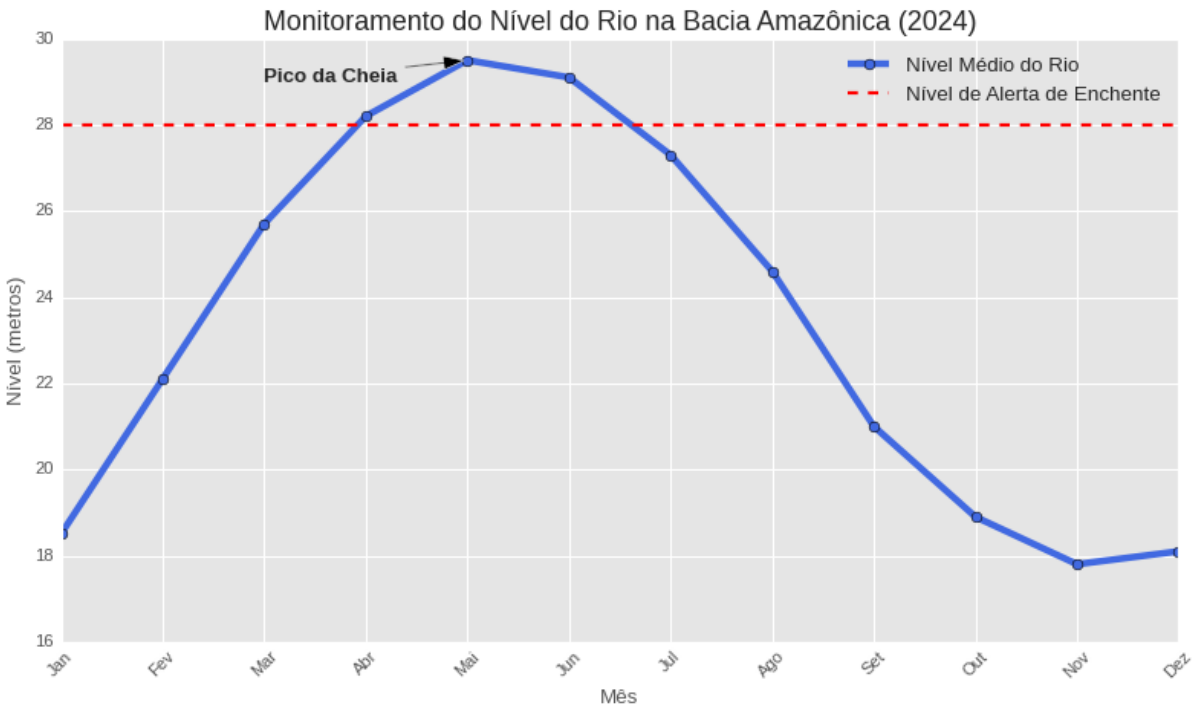

Figura 3.8: Gráfico de linhas com customizações específicas para uma certa finalidade

Comparando Gráficos Lado a Lado com Subplots

E se quisermos investigar a *causa* da cheia do rio? A suspeita principal na Amazônia é a chuva. Para visualizar essa relação de causa e efeito, a melhor abordagem é colocar os gráficos de chuva e de nível do rio juntos na mesma figura, usando **subplots**.

Caso Prático

Um hidrólogo quer demonstrar visualmente como o ciclo de chuvas impacta o nível do rio, destacando o atraso entre o pico da chuva e o pico da cheia. Para isso, precisamos criar uma única figura com dois gráficos: um de barras para a precipitação mensal e um de linha para o nível do rio.

Copie e Teste!

```
import pandas as pd
import matplotlib.pyplot as plt

plt.style.use('seaborn-v0_8-whitegrid')

# --- DADOS ---
dados = {
    'mes': ['Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun', 'Jul', 'Ago'
   , 'Set', 'Out', 'Nov', 'Dez'],
    'nivel_m': [18.5, 22.1, 25.7, 28.2, 29.5, 29.1, 27.3, 24.6,
   21.0, 18.9, 17.8, 18.1],
    'precipitacao_mm': [310, 300, 320, 250, 180, 90, 60, 40, 70,
   120, 150, 210]
}
df = pd.DataFrame(dados)

# --- CRIAÇÃO DOS SUBPLOTS ---
# Figura com 2 linhas e 1 coluna. 'sharex=True' alinha os meses.
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10, 8), sharex
   =True)

# Adiciona um título geral para a figura inteira
fig.suptitle('Análise Sazonal: Chuva vs. Nível do Rio', fontsize
   =16, fontweight='bold')

# PLOT 1: Gráfico de Barras da Chuva (no 1º eixo: axes[0])
axes[0].bar(df['mes'], df['precipitacao_mm'], color='royalblue',
   label='Chuva')
axes[0].set_ylabel('Precipitação (mm)')
axes[0].legend()

# PLOT 2: Gráfico de Linha do Nível do Rio (no 2º eixo: axes[1])
axes[1].plot(df['mes'], df['nivel_m'], color='darkorange', marker=
   'o', label='Nível do Rio')
axes[1].set_ylabel('Nível (metros)')
axes[1].set_xlabel('Mês')
axes[1].legend()

# Ajustes finais
plt.tight_layout(rect=[0, 0, 1, 0.96]) # Espaço para o subtitle
plt.show()
```

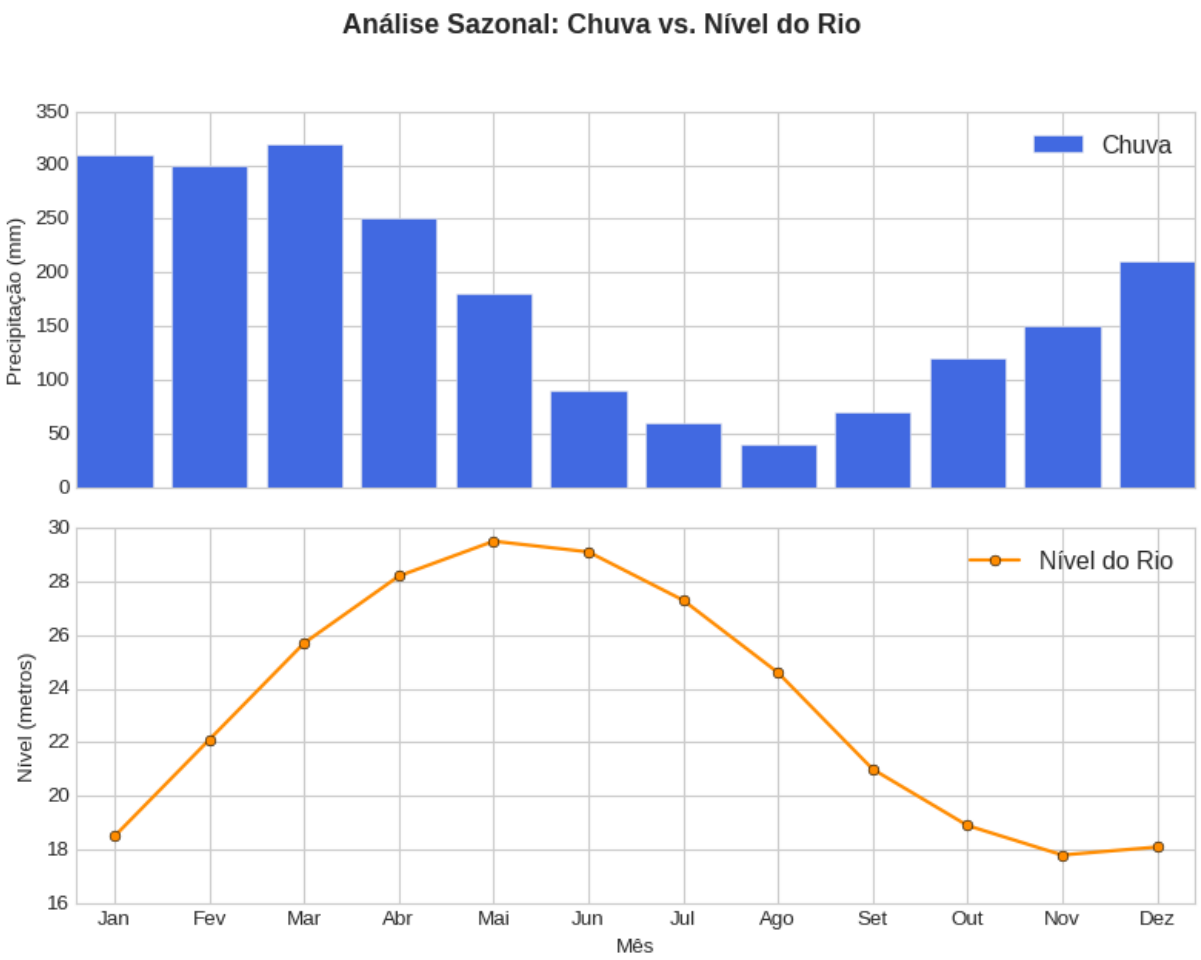

Figura 3.9: Gráficos concatenados através do subplot

A História que os Subplots Contam
Ao colocar os gráficos juntos, a hipótese do hidrólogo fica visualmente clara:

1. **Causa (Chuva):** O gráfico de barras superior mostra que o período mais chuvoso se concentra entre Janeiro e Abril.

2. **Efeito (Nível do Rio):** O gráfico de linha inferior mostra que o rio continua a subir mesmo depois que as chuvas diminuem, atingindo seu pico em Maio/Junho.

3. **A Descoberta (O Atraso):** A visualização confirma o ”lag”ou atraso. Leva tempo para que a água das chuvas que caem em toda a bacia hidrográfica chegue ao rio principal e cause o pico da cheia.

Esta é a verdadeira força da customização avançada: não se trata apenas de fazer um gráfico, mas de **projetar uma visualização** que responda a uma pergunta específica e revele padrões complexos.

Alternativas Visuais para Dados Categóricos

Vamos verificar como os mesmos dados podem ser apresentados de formas alternativa para contar histórias diferentes, usando nosso exemplo da contagem de peixes `df_peixes`.

Barras Horizontais (`barh`): Para Clareza em Rótulos Longos
Quando os nomes das categorias são longos, um gráfico de barras horizontais é a melhor escolha para evitar a sobreposição de texto e facilitar a leitura. Nesse exemplo vamos agrupar pela média de peixes (ao invés do total) em diferentes locais (ao invés da espécie).

Copie e Teste!

```
# 1. Agrupar os dados para obter a média por local e ordenar as
   barras de forma decrescente (maior para o menor)
contagem_por_local = df_peixes.groupby('local')['quantidade'].mean
   ().sort_values()

# 2. Criar o gráfico de barras horizontal com os dados
contagem_por_local.plot(kind='barh', color=['skyblue', 'salmon', '
   lightgreen', 'gold'])

# 3. Customizar e exibir o gráfico
plt.title('Média de Peixes por Local')
plt.xlabel('Quantidade Média Observada')
plt.ylabel('Local')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

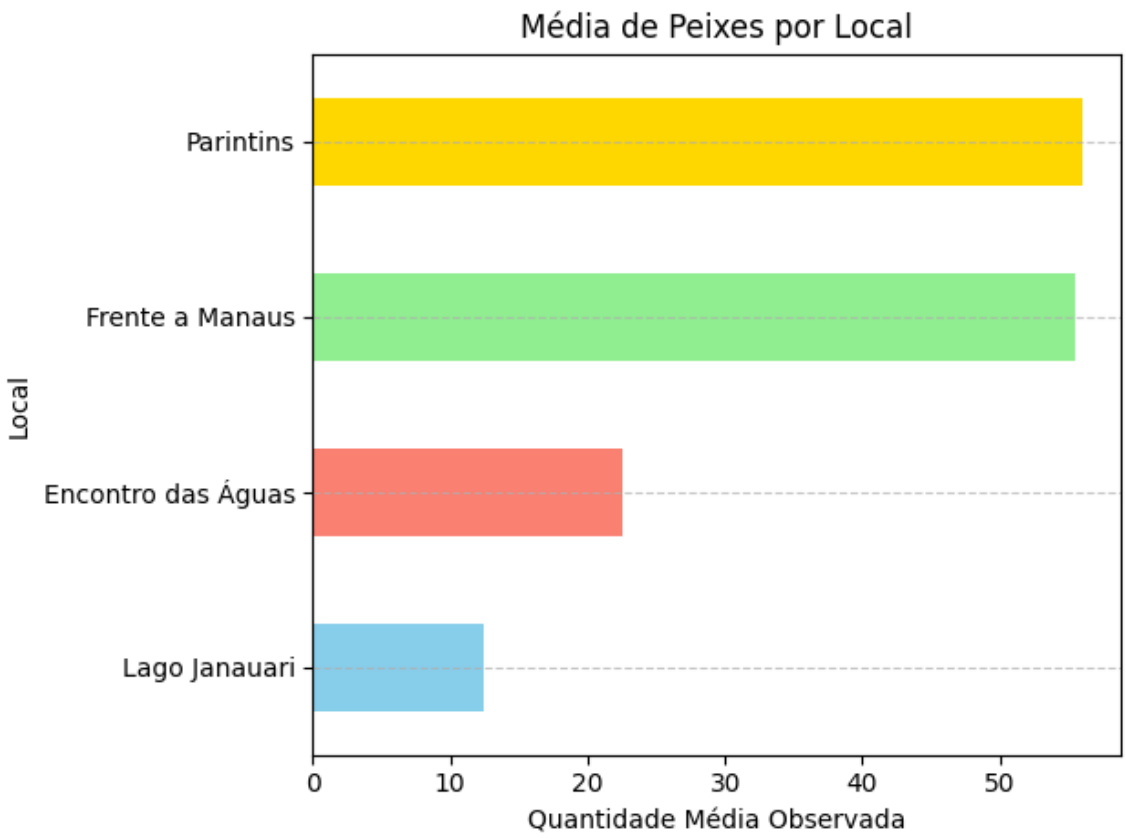

Figura 3.10: Gráficos de barras horizontais agrupado por local

Gráfico de Pizza (`pie`): Para Mostrar Proporções de um Todo
Se o objetivo é verificar a **porcentagem** de cada categoria em relação ao total, um gráfico de pizza pode ser uma alternativa mais indicada.

Copie e Teste!

```
contagem_por_local.plot(
    kind='pie', # Estilo do gráfico
    y='quantidade', # Quais dados serão usados nas porcentagens
    autopct='%1.1f%%', # Mostra as porcentagens
    figsize=(6, 6), # Tamanho do gráfico
    legend=False, # Nesse caso é desnecessário
    labels=contagem_por_local.index, # Rótulos dos locais
    colors=['skyblue', 'salmon', 'lightgreen', 'gold']
)
plt.title('Composição Relativa da Média de Peixes Coletados')
plt.ylabel('') # Remove o rótulo 'quantidade' que é desnecessário
plt.show()
```

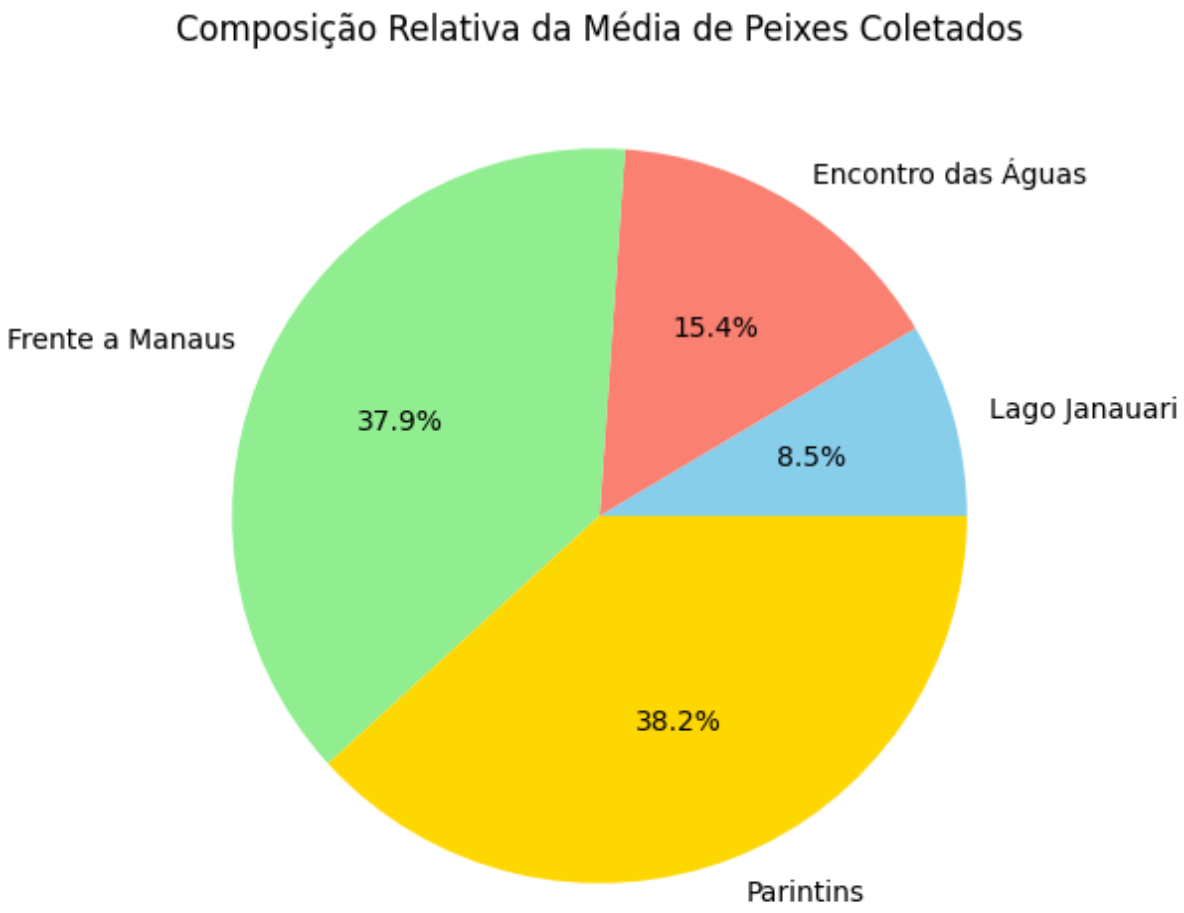

Figura 3.11: Gráficos de pizza agrupado por local

Fique Alerta!

Use gráficos de pizza com moderação. Eles funcionam bem para poucas categorias (geralmente menos de 6). Quando há muitas fatias, a comparação se torna difícil e um gráfico de barras é quase sempre uma escolha melhor e mais clara.

Dominar essas técnicas de customização eleva seu trabalho de simples plotagem de dados para uma verdadeira **comunicação visual de insights**, uma habilidade essencial para qualquer analista de dados.

3.4.3 Gráficos Estatísticos

Nos tópicos anteriores, aprendemos a usar gráficos de linhas para ver tendências e gráficos de barras para comparar categorias. Essas são ferramentas fantásticas, como um mapa e uma régua para nossas expedições de dados. No entanto, para entender a história completa, um explorador precisa de ferramentas mais avançadas, como uma lupa para ver a composição do solo ou um binóculo para enxergar padrões à distância.

É aqui que entram os **gráficos estatísticos**. Eles não mostram apenas os dados; eles revelam a *estrutura interna* das informações: sua **distribuição**, sua **dispersão** e as **relações** entre diferentes variáveis. Se os gráficos de barras e linhas respondem ”quanto?”e ”quando?”, os gráficos estatísticos respondem ”com que frequência?”, ”quão espalhados?”e ”eles estão conectados?”.

Nesta seção, vamos aprofundar nossa análise com as seguintes ferramentas:

- **Histograma:** Para entender a frequência e a distribuição dos nossos dados.
- **Boxplot:** Para resumir a dispersão dos dados e comparar grupos de forma eficaz.
- **Diagrama de Dispersão (Scatter Plot):** Para investigar a relação entre duas variáveis numéricas.

Conjunto de Dados: A Safra de Castanha-do-Pará

Caso Prático

Imagine que somos analistas de dados em uma cooperativa de produtores de castanha-do-pará (*Bertholletia excelsa*) na Amazônia. A cooperativa coletou dados de centenas de árvores para entender melhor o que influencia uma boa colheita. Os dados estão no arquivo `safra_castanha.csv` e contêm as seguintes informações para cada árvore:

- **id_arvore:** Um identificador único para cada castanheira.
- **regiao:** A região de coleta (ex: ’Reserva Tapajós’, ’Floresta Jari’).
- **diametro_cm:** O diâmetro do tronco da árvore em centímetros, uma medida que ajuda a estimar a idade e a saúde da árvore.
- **producao_kg:** A quantidade total de castanhas coletadas daquela árvore na última safra, em quilogramas.

Nosso objetivo é usar gráficos estatísticos para transformar essa tabela de números em insights que ajudem a cooperativa a melhorar suas práticas de manejo sustentável.

O primeiro passo para iniciar nossa exploração e análise é carregar e inspecionar nossos dados referentes a safra de castanha-do-pará.

Copie e Teste!

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Para garantir que nossos exemplos sejam reprodutíveis, vamos
   criar os dados
np.random.seed(42) # Semente para reprodutibilidade
dados = {
    'id_arvore': range(1, 101),
    'regiao': np.random.choice(['Reserva Tapajós', 'Floresta Jari'
   ], 100),
    'diametro_cm': np.random.normal(loc=120, scale=25, size=100),
    'producao_kg': np.nan # Vamos preencher com base no diâmetro
}
df_safra = pd.DataFrame(dados)

# Criando uma relação realista entre diâmetro e produção com um
   pouco de ruído
df_safra['producao_kg'] = (df_safra['diametro_cm'] * 0.45) + np.
   random.normal(loc=0, scale=10, size=100)
# Garantindo que não haja produção negativa
df_safra['producao_kg'] = df_safra['producao_kg'].clip(lower=5)

print("--- Primeiras 5 linhas do nosso conjunto de dados ---")
print(df_safra.head())

print("\n--- Informações gerais do DataFrame ---")
df_safra.info()
```

Saída Esperada

```
--- Primeiras 5 linhas do nosso conjunto de dados ---
   id_arvore           regiao  diametro_cm  producao_kg
0          1  Reserva Tapajós   132.410131    62.887648
1          2    Floresta Jari   114.721473    67.877897
2          3  Reserva Tapajós   128.535815    59.623193
3          4  Reserva Tapajós   155.132333    74.288219
4          5  Reserva Tapajós   111.230291    43.053155

--- Informações gerais do DataFrame ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 4 columns):
```

```
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   id_arvore    100 non-null    int64
 1   regiao       100 non-null    object
 2   diametro_cm  100 non-null    float64
 3   producao_kg  100 non-null    float64
dtypes: float64(2), int64(1), object(1)
memory usage: 3.2+ KB
```

Histograma: A Frequência dos Dados

O primeiro passo para entender uma variável numérica é olhar sua **distribuição**. Um histograma é a ferramenta perfeita para isso. Ele agrupa os dados em ”faixas”(ou *bins*) de igual tamanho e conta quantos valores caem em cada faixa. O resultado é um gráfico de barras que mostra onde os dados estão mais concentrados.

Caso Prático

A cooperativa quer saber qual é a faixa de produção mais comum entre as castanheiras. Existem muitas árvores de baixa produção e poucas de alta produção, ou a maioria se concentra em um valor médio?

Copie e Teste!

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid')

# Criando a figura e os eixos
fig, ax = plt.subplots(figsize=(10, 6))

# Plotando o histograma da coluna 'producao_kg' com 20 intervalos
ax.hist(df_safra['producao_kg'], bins=20, color='forestgreen',
   edgecolor='black')

# Customizando o gráfico
ax.set_title('Distribuição da Produção de Castanha-do-Pará',
   fontsize=16)
ax.set_xlabel('Produção por Árvore (kg)', fontsize=12)
ax.set_ylabel('Frequência (N° de Árvores)', fontsize=12)
plt.show()
```

O histograma 3.12 nos mostra uma imagem interessante que se assemelha a um sino, conhecida como **distribuição normal** ou ”curva de sino”. Isso nos diz que:

- A maioria das árvores tem uma produção concentrada em torno do centro do gráfico (provavelmente entre 40 e 60 kg).
- Árvores com produção muito baixa ou muito alta são menos frequentes, como mostram as barras mais curtas nas extremidades.

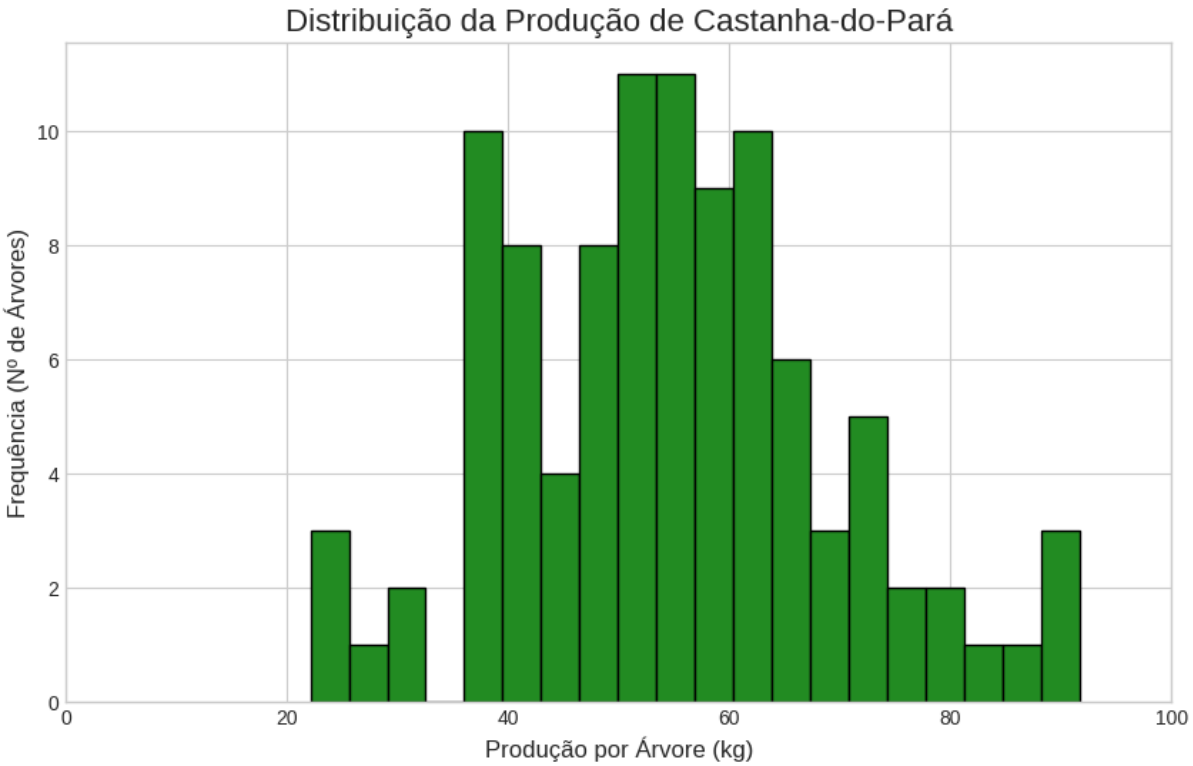

Figura 3.12: Histograma mostrando a distribuição da produção de castanhas.

Essa ”forma”dos dados é um *insight* crucial que não conseguiríamos ver apenas com a média ou a mediana. Ela nos informa que a produção das castanheiras segue um padrão natural e previsível.

Boxplot: Um Raio-X da Dispersão

Enquanto o histograma mostra a forma da distribuição, o **boxplot** (ou diagrama de caixa) nos dá um resumo estatístico em uma única imagem. Ele é excelente para entender a **dispersão** dos dados e para **comparar** distribuições entre diferentes categorias. Um boxplot mostra cinco informações importantes:

- **Mediana (linha no meio da caixa):** O valor central dos dados.
- **Primeiro Quartil (Q1 - base da caixa):** 25% dos dados estão abaixo deste valor.
- **Terceiro Quartil (Q3 - topo da caixa):** 25% dos dados estão acima deste valor.
- **”Bigodes”(whiskers):** Mostram o alcance da maioria dos dados.
- **Outliers:** Valores que são considerados atipicamente altos ou baixos.

Caso Prático

A cooperativa quer comparar a produtividade das árvores entre as duas regiões de coleta: ’Reserva Tapajós’ e ’Floresta Jari’. Qual região tem a maior produção média? E qual delas tem a produção mais consistente (menos variável)? Um boxplot comparativo é a ferramenta ideal para responder a isso.

Copie e Teste!

```
import pandas as pd
import matplotlib.pyplot as plt

# O Pandas tem uma integração direta para criar boxplots por
    categoria
fig, ax = plt.subplots(figsize=(10, 7))

# Usamos o método .boxplot() do DataFrame
# 'column' é a variável que queremos analisar
# 'by' é a categoria pela qual queremos agrupar
df_safra.boxplot(column='producao_kg', by='regiao', ax=ax, grid=
    False)

# Customizando o gráfico
ax.set_title('Comparação da Produção de Castanha por Região',
    fontsize=16)
ax.set_ylabel('Produção por Árvore (kg)', fontsize=12)
ax.set_xlabel('Região de Coleta', fontsize=12)
fig.suptitle('') # Remove o título automático do pandas

plt.show()
```

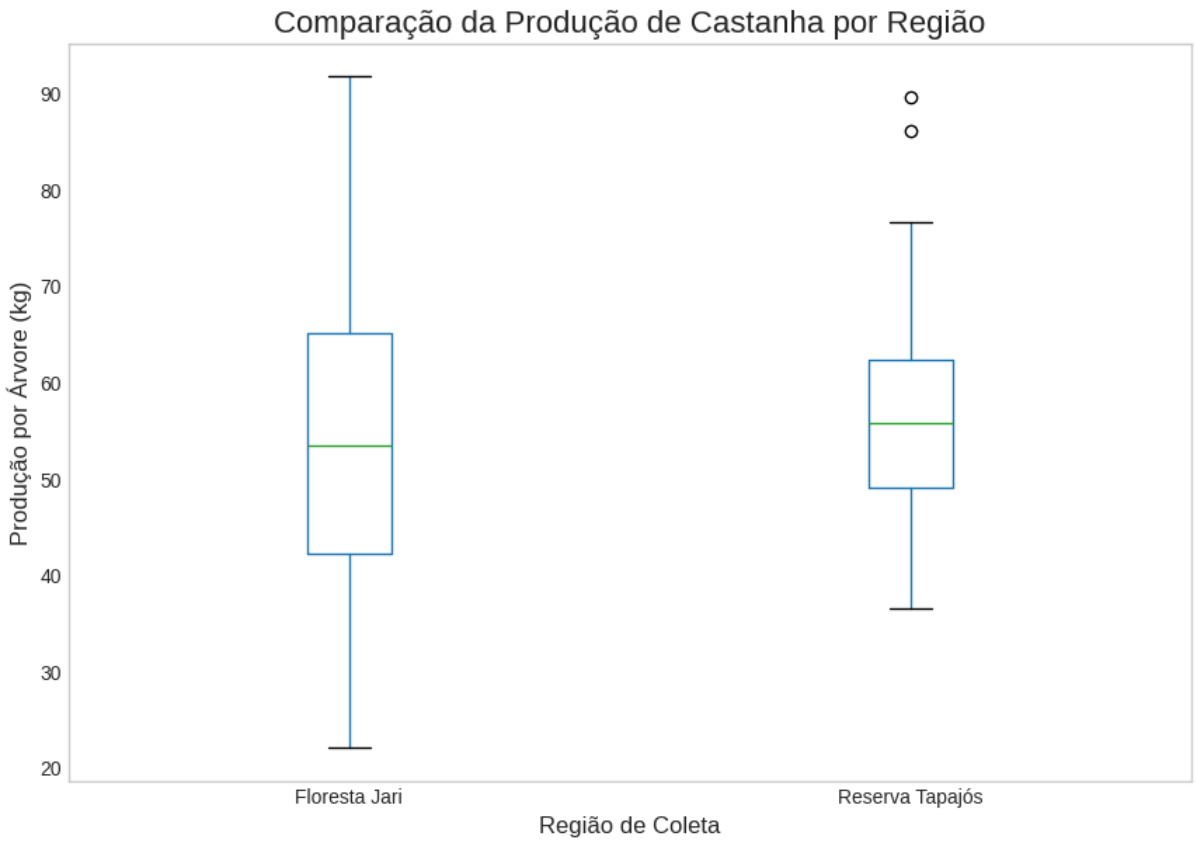

Figura 3.13: Boxplot comparando a produção de castanhas entre duas regiões.

O boxplot comparativo 3.13 nos permite tirar conclusões imediatas:

- A **Floresta Jari** parece ter uma mediana de produção ligeiramente mais alta (a linha dentro da caixa está mais para cima).
- A **Reserva Tapajós** apresenta uma dispersão maior (a caixa e os "bigodes"são mais longos), o que sugere que a produção nessa área é mais variável, com árvores produzindo tanto muito pouco quanto muito.

Diagrama de Dispersão: Investigando Relações

Até agora, analisamos uma variável de cada vez. Mas a pergunta mais interessante na ciência de dados é: **uma variável afeta a outra?** Para investigar a relação entre duas variáveis numéricas, usamos o **diagrama de dispersão** (*scatter plot*).

Ele plota cada observação como um ponto em um gráfico de dois eixos. Se os pontos formam um padrão (como uma linha subindo ou descendo), isso sugere uma correlação entre as variáveis.

Caso Prático

A sabedoria popular entre os coletores de castanha diz que "árvore mais grossa dá mais fruto". A cooperativa quer verificar se os dados confirmam essa hipótese. Existe uma relação entre o diâmetro da castanheira (`diametro_cm`) e a sua produção (`producao_kg`)?

Copie e Teste!

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(10, 6))

# Criando o diagrama de dispersão
# O eixo x será o diâmetro, e o eixo y, a produção
ax.scatter(x=df_safra['diametro_cm'], y=df_safra['producao_kg'],
   alpha=0.7, color='saddlebrown')

# Customizando
ax.set_title('Relação entre Diâmetro e Produção da Castanheira',
   fontsize=16)
ax.set_xlabel('Diâmetro do Tronco (cm)', fontsize=12)
ax.set_ylabel('Produção de Castanhas (kg)', fontsize=12)

plt.show()
```

O diagrama de dispersão 3.14 confirma a sabedoria popular! Os pontos formam um padrão claro que sobe da esquerda para a direita, o que indica uma **correlação positiva**. De forma geral, é possível inferir com algum grau de confiança que quanto maior o diâmetro da árvore, maior é a sua produção de castanhas.

Essa é uma das descobertas mais importantes que podemos fazer, pois nos permite usar a variável diâmetro (que é fácil de medir) para prever a produção, uma ideia que é a base para modelos de *Machine Learning*. Como ressaltado por McKinney (2018), a capacidade de visualizar relações é fundamental para guiar a análise de dados.

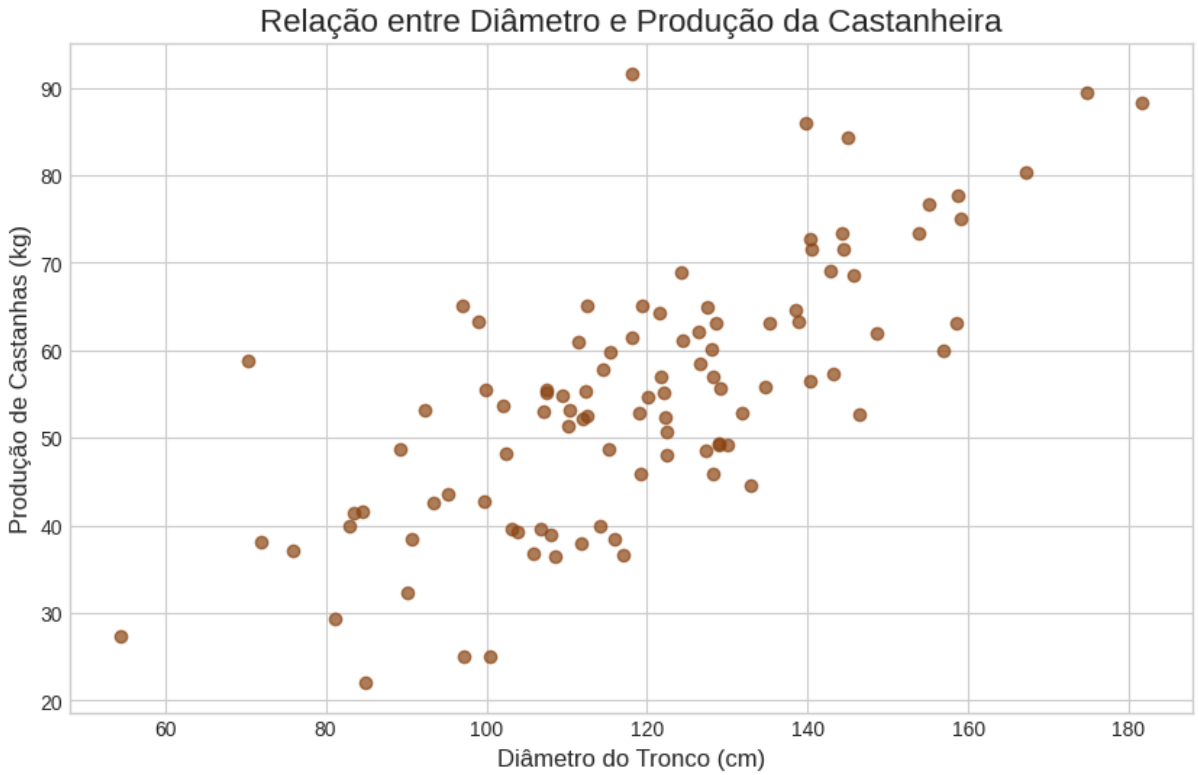

Figura 3.14: Diagrama de dispersão mostrando a relação entre diâmetro e produção.

3.5 Considerações do Módulo 3

Conhecendo um pouco mais!

Materiais Práticos do Módulo 3
Lembre-se que todos os scripts, datasets e o notebook para este módulo estão disponíveis em nosso repositório no GitHub. Para Instruções detalhadas sobre como utilizar os materiais, consulte a subseção 1.5 Considerações do Módulo 1

- **Scripts e Datasets do Módulo 3:**
  `https://github.com/CITHA-AM/Python/tree/main/Modulo%203`
- **Notebook do Módulo 2 (Colab):**
  `https://colab.research.google.com/github/CITHA-AM/Python/blob/main/Modulo%203.ipynb`

Parabéns por completar o Módulo 3! Você acaba de realizar uma jornada transformadora, saindo do mundo da programação estruturada para entrar no fascinante universo da Análise de Dados. Nos módulos anteriores, você construiu as fundações, aprendendo a organizar o código e a gerenciar informações. Agora, você aprendeu a dar sentido a essas informações, transformando dados brutos em conhecimento valioso.

Neste capítulo, você foi apresentado a duas das mais poderosas ferramentas do ecossistema Python: **Pandas** e **Matplotlib**. Com o Pandas, você aprendeu a organizar o caos dos dados brutos em `DataFrames` limpos e estruturados, uma habilidade essencial para qualquer

tipo de análise. Você dominou as etapas cruciais do fluxo de trabalho de um analista:

- **Inspeção:** Utilizando métodos como `.info()` e `.head()` para entender a estrutura e a saúde dos seus dados.
- **Limpeza e Manipulação:** Aprendendo a selecionar, filtrar e, crucialmente, a lidar com dados ausentes, garantindo a confiabilidade da sua análise.
- **Análise e Agregação:** Desvendando padrões através de estatísticas descritivas e, mais importante, com a poderosa técnica de `groupby` para comparar diferentes segmentos dos seus dados.

Em seguida, com o Matplotlib, você aprendeu a arte de contar histórias com dados. Você viu como transformar as tabelas e os resultados de suas análises em visualizações claras e impactantes, desde gráficos de linhas que mostram tendências ao longo do tempo até gráficos de barras que comparam categorias. Você também explorou gráficos estatísticos como histogramas, boxplots e diagramas de dispersão, que revelam a estrutura interna das informações, como suas distribuições e correlações.

As habilidades que você adquiriu neste módulo são a base da ciência de dados. Você não está mais apenas escrevendo código que executa tarefas; você está programando para fazer perguntas, investigar hipóteses e comunicar descobertas de forma eficaz. Você aprendeu a extrair significado dos dados, uma competência essencial no mundo tecnológico atual.

Com esta base sólida em manipulação e visualização de dados, você está mais do que preparado para o próximo passo. No Módulo 4, aplicaremos todo esse conhecimento em projetos práticos, enfrentando desafios do mundo real. Você usará o poder do Pandas e do Matplotlib para realizar análises completas, consolidando sua jornada e transformando-o em um solucionador de problemas orientado a dados. Continue com o excelente trabalho!

Capítulo 4

Projetos Práticos Aplicados

Iniciando o diálogo...

Parabéns por chegar ao capítulo decisivo da nossa jornada! Se nos módulos anteriores construímos nossa ”casa do conhecimento”— aprendendo o idioma da programação com Python, erguendo as paredes com funções e arquivos, e organizando os móveis com Pandas e Matplotlib — agora é o momento de usá-la para um propósito real. Este é o capítulo da ”mão na massa”, onde a teoria se transforma em solução.

Até agora, você se dedicou a montar um poderoso arsenal de ferramentas. No Módulo 1, você aprendeu a sintaxe e a lógica fundamental do Python. No Módulo 2, descobriu como organizar seu código de forma eficiente com funções e a dar memória aos seus programas com a manipulação de arquivos. Em seguida, no Módulo 3, você mergulhou no universo da análise de dados, aprendendo a usar o Pandas para transformar dados brutos em tabelas organizadas e o Matplotlib para converter números em histórias visuais.

O verdadeiro valor de um programador ou analista de dados, no entanto, não reside apenas em conhecer as ferramentas, mas em saber aplicá-las para resolver problemas concretos. Este capítulo foi projetado para ser exatamente essa ponte entre o conhecimento e a prática. Aqui, você enfrentará dois desafios realistas e de grande relevância para o contexto amazônico:

- Análise de Dados Agrícolas: Ajudar uma cooperativa a otimizar suas práticas de cultivo, usando dados para entender produtividade e uso de recursos de diferentes culturas.
- Análise de Piscicultura: Assumir o papel de consultor para um pequeno produtor, analisando seus dados de produção para recomendar o peixe mais rentável para sua criação.

Em cada experimento, você aplicará todo o fluxo de trabalho de um projeto de análise de dados: desde a importação e limpeza, passando pela análise e agregação, até a visualização e interpretação dos resultados. Ao final deste módulo, você não terá apenas concluído os exercícios; você terá construído soluções de ponta a ponta, consolidando sua confiança para enfrentar seus próprios desafios com dados no futuro.

4.1 Experimento 1: Análise de Dados Agrícolas

Imagine que você é um analista de dados contratado por uma cooperativa agrícola na Amazônia. Com a crescente necessidade de otimizar a produção e usar os recursos naturais de forma sustentável, a gestão precisa de ajuda para tomar decisões baseadas em evidências.

Sua missão é analisar um conjunto de dados de safras passadas para responder a perguntas cruciais:

- Qual cultivo tem sido o mais produtivo?
- Qual deles é mais eficiente no uso da água?
- Como a produtividade tem evoluído ao longo dos anos?

Vamos usar o poder do Python para transformar esses dados brutos em insights valiosos.

4.1.1 Análise do Problema e dos Dados

O primeiro passo em qualquer projeto de análise de dados é entender o problema e conhecer os dados disponíveis. Nosso objetivo é realizar uma análise exploratória em um conjunto de dados agrícolas para extrair informações que possam guiar o planejamento de futuras safras. O conjunto de dados, contido no arquivo `dados_agricolas.csv`, possui a seguinte estrutura:

- **tipo_cultivo:** O nome do cultivo (ex: Soja, Trigo, Milho).
- **produtividade:** A produção registrada em toneladas por hectare.
- **uso_agua:** O consumo de água em litros por hectare.
- **custo_medio:** O custo médio de produção por hectare.
- **ano:** O ano em que o registro foi feito.

Conhecendo um pouco mais!

Acessando os Dados do Experimento
Para executar os scripts dos experimentos, será necessário carregar os dados à partir do arquivo `dados_agricolas.csv`, diretamente de um repositório público no GitHub. Esta é uma prática padrão em ciência de dados, garantindo que qualquer pessoa possa reproduzir a análise.

Caso Prático

Passo 1: Leitura e Inspeção Inicial dos Dados via URL
Raciocínio: Antes de qualquer cálculo, precisamos carregar os dados para nosso ambiente. A função `pd.read_csv()` do Pandas pode ler dados diretamente de uma URL. Após o carregamento, faremos a inspeção inicial com `.head()` e `.info()` para garantir que os dados foram recebidos e interpretados corretamente.

Copie e Teste!

```
import pandas as pd

# URL para acesso dos dados brutos no GitHub
url = 'https://raw.githubusercontent.com/CITHA-AM/Python/refs/heads/main/Modulo%204/dados_agricolas.csv'
```

```python
# Passo 1: Leitura e Inspeção Inicial dos Dados
# Carregando os dados diretamente da URL
df = pd.read_csv(url)

print("--- Dados carregados com sucesso a partir da URL! ---")
# Visualizando as 5 primeiras linhas
print(df.head())

# Obtendo informações gerais do DataFrame
print("\n--- Informações e Estrutura do DataFrame ---")
df.info()
```

Saída Esperada

```
--- Dados carregados com sucesso a partir da URL! ---
      tipo_cultivo  produtividade  uso_agua  custo  ano
0             Soja           6.82       605    947  2012
1            Trigo           9.67       800   1220  2023
2            Trigo          11.45       426    843  2021
3   Cana-de-acucar           5.25       808    959  2023
4            Trigo           7.48       686    841  2020

--- Informações e Estrutura do DataFrame ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   tipo_cultivo   100 non-null    object
 1   produtividade  100 non-null    float64
 2   uso_agua       100 non-null    int64
 3   custo_medio    100 non-null    int64
 4   ano            100 non-null    int64
dtypes: float64(1), int64(3), object(1)
memory usage: 4.0+ KB
```

Interpretação: A inspeção inicial mostra que os 100 registros foram carregados corretamente a partir da URL. Não há valores ausentes e os tipos de dados foram identificados corretamente pelo Pandas, o que nos permite prosseguir com a análise.

4.1.2 Implementação da Solução em Python

Com os dados carregados e inspecionados, nosso próximo passo é executar as análises necessárias para responder às perguntas da cooperativa. Nesta seção, vamos implementar a solução para cada um dos objetivos analíticos, explicando o raciocínio e o código utilizado.

Caso Prático

Passo 2: Calcular a Produtividade Média por Cultivo
Raciocínio: Para descobrir a produtividade média de cada tipo de cultivo, precisamos agrupar nosso DataFrame pela coluna `tipo_cultivo`. Após agrupar, podemos selecionar a coluna `produtividade` e aplicar a função de agregação `.mean()` para calcular a média para cada grupo. Usaremos `.reset_index()` para transformar o resultado de volta em um DataFrame limpo.

Copie e Teste!

```
# Passo 2: Tabela de produtividade média por produto
print("--- 2. Produtividade Média por Tipo de Cultivo ---")
produtividade_media = df.groupby('tipo_cultivo')['produtividade'].
    mean().reset_index()
print(produtividade_media.round(2))
```

Saída Esperada

```
--- 2. Produtividade Média por Tipo de Cultivo ---
      tipo_cultivo  produtividade
0            Arroz           6.38
1   Cana-de-acucar           6.41
2            Milho           7.71
3             Soja           6.90
4            Trigo           8.44
```

Interpretação: A tabela nos mostra um resumo claro da performance média de cada cultura. Podemos observar que o **Trigo** foi, em média, o cultivo mais produtivo, enquanto o **Arroz** apresentou la menor produtividade média entre os registros analisados.

Caso Prático

Passo 3: Identificar o Cultivo com Maior Produtividade
Raciocínio: Já temos o DataFrame com a produtividade média de cada cultivo, criado no passo anterior. Agora, para encontrar o cultivo com o maior valor, podemos usar o método `.idxmax()` na coluna `produtividade` para descobrir o índice da linha com o maior valor. Em seguida, usamos `.loc[]` para selecionar e exibir essa linha inteira.

Copie e Teste!

```
# Passo 3: Apresentar o produto com maior produtividade
print("\n--- 3. Cultivo com Maior Produtividade Média ---")
cultivo_mais_produtivo = produtividade_media.loc[
    produtividade_media['produtividade'].idxmax()]
print(cultivo_mais_produtivo)
```

Saída Esperada

```
--- 3. Cultivo com Maior Produtividade Média ---
tipo_cultivo      Trigo
produtividade      8.44
Name: 4, dtype: object
```

Interpretação: A análise confirma que o **Trigo**, com uma produtividade média de 8.44 toneladas por hectare, foi o cultivo de maior performance no período analisado.

Caso Prático

Passo 4: Identificar o Cultivo com Menor Uso Médio de Água
Raciocínio: A lógica é muito similar à do passo anterior, mas agora focada na eficiência hídrica. Primeiro, agrupamos os dados por `tipo_cultivo` e calculamos a média da coluna `uso_agua`. Depois, usamos `.idxmin()` para encontrar o índice do cultivo com o menor valor médio de consumo de água.

Copie e Teste!

```
# Passo 4: Apresentar o produto com menor uso médio de água
print("\n--- 4. Cultivo com Menor Uso Médio de Água ---")
uso_agua_medio = df.groupby('tipo_cultivo')['uso_agua'].mean().
    reset_index()
cultivo_menor_uso_agua = uso_agua_medio.loc[uso_agua_medio['
    uso_agua'].idxmin()]
print(cultivo_menor_uso_agua)
```

Saída Esperada

```
--- 4. Cultivo com Menor Uso Médio de Água ---
tipo_cultivo      Soja
uso_agua        649.38
Name: 3, dtype: object
```

Interpretação: Em termos de sustentabilidade e uso de recursos, a **Soja** se destacou como o cultivo mais eficiente, apresentando o menor consumo médio de água por hectare.

4.1.3 Apresentação dos Resultados

Após a análise quantitativa, a etapa final do nosso experimento é transformar nossas descobertas em uma visualização de dados eficaz. Gráficos são ferramentas poderosas para comunicar tendências que podem ser difíceis de discernir em tabelas. Nesta seção, vamos construir um gráfico de linhas para visualizar como a produtividade de cada cultivo evoluiu ao longo dos anos, passando por um processo de diagnóstico e refinamento, que é muito comum em projetos reais.

Caso Prático

Passo 5, Parte A: Primeira Tentativa de Visualização
Raciocínio: Nossa primeira abordagem será agrupar os dados por ano e cultivo para obter a produtividade média e, em seguida, plotar o resultado. Isso nos ajudará a ter uma primeira impressão visual das tendências e a identificar possíveis problemas nos dados.

Copie e Teste!

```
import matplotlib.pyplot as plt

# Passo 5:Gerando gráfico da produtividade
# Agrupando para obter a produtividade média por ano e por cultivo
df_agrupado = df.groupby(['ano', 'tipo_cultivo'])['produtividade'
    ].mean().unstack()

# Criando o gráfico
fig, ax = plt.subplots(figsize=(14, 8))
df_agrupado.plot(kind='line', marker='o', ax=ax)

# Customizando o gráfico
ax.set_title('Produtividade Média por Cultivo', fontsize=16)
ax.set_xlabel('Ano', fontsize=12)
ax.set_ylabel('Produtividade Média (toneladas/hectare)', fontsize
    =12)
ax.legend(title='Tipo de Cultivo')
plt.show()
```

Análise do Problema Visual: Observando o gráfico 4.1, notamos um problema imediato, já que as linhas estão descontínuas. Isso ocorre porque nosso conjunto de dados não possui registros para todos os anos para todos os cultivos. O Pandas preenche essas lacunas com valores nulos (`NaN`), e o Matplotlib interrompe a linha ao encontrá-los. Essa visualização, embora precisa, não é eficaz para comparar tendências.

Fique Alerta!

Qual Critério Usar: Média ou Soma?
Nossa análise agrupa os dados pela **média** da produtividade. Mas por que não usar a **soma**? A escolha depende da pergunta.

- **Média (‘.mean()‘):** Responde ”Qual a produtividade *típica* ou *eficiência* de um cultivo?”. É ideal para comparar o desempenho entre diferentes cultivos de forma justa.
- **Soma (‘.sum()‘):** Responde ”Qual o *volume total* de produção registrado?”. Seria útil se os dados representassem a produção total de uma fazenda, mas pode ser enganosa em dados de amostragem.

Para o objetivo da cooperativa de comparar a **eficiência** dos cultivos, a **média** é a métrica mais correta e estratégica.

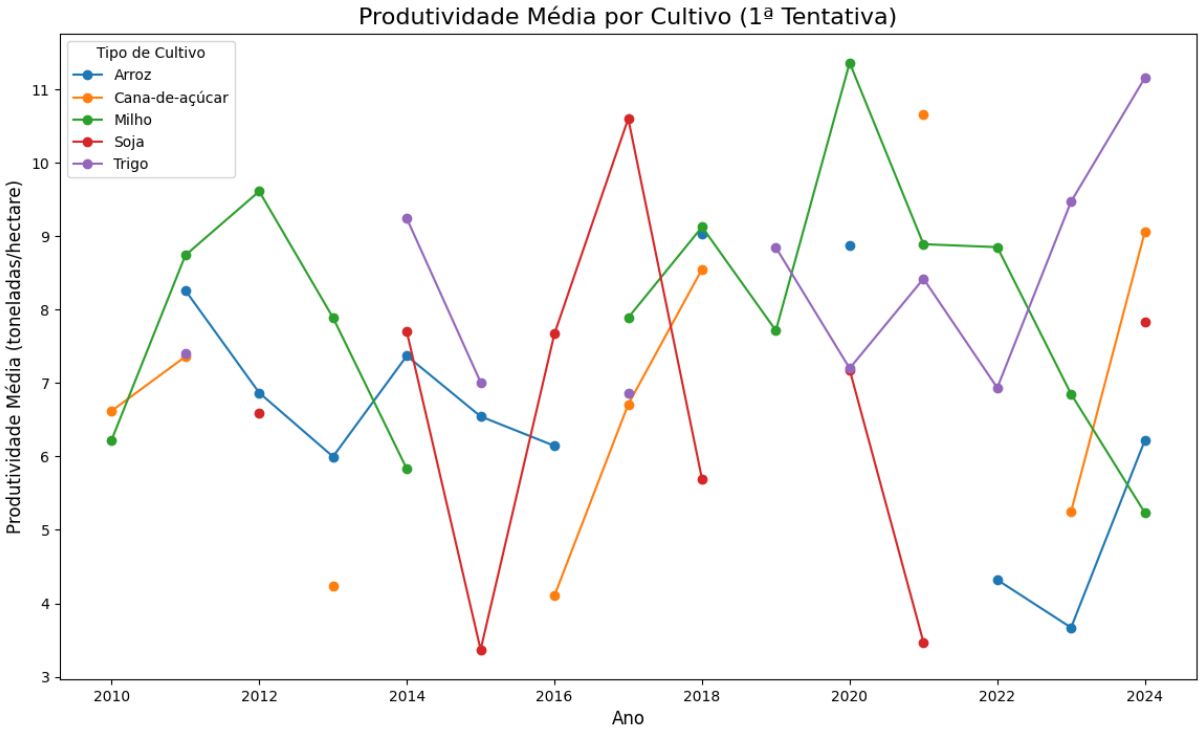

Figura 4.1: Evoluação temporal da produtividade média por tipo de cultivo (dados faltantes)

Caso Prático

Passo 5, Parte B: Corrigindo o Gráfico com Interpolação
Raciocínio: Para criar uma visualização de tendência mais clara, podemos preencher as lacunas nos dados. A técnica ideal para isso é a **interpolação linear**, que estima os valores ausentes traçando uma linha reta entre os pontos de dados conhecidos, resultando em linhas contínuas que facilitam a comparação.

Copie e Teste!

```
# Interpolando os dados para preencher anos faltantes
df_interpolado = df_agrupado.interpolate(method='linear')

# Criando o gráfico com os dados corrigidos
fig, ax = plt.subplots(figsize=(14, 8))
df_interpolado.plot(kind='line', marker='o', ax=ax)

# Customizando o gráfico final
ax.set_title('Produtividade Média por Cultivo ao Longo dos Anos (
    com Interpolação)', fontsize=16)
ax.set_xlabel('Ano', fontsize=12)
ax.set_ylabel('Produtividade Média (toneladas/hectare)', fontsize
    =12)
ax.legend(title='Tipo de Cultivo')
ax.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

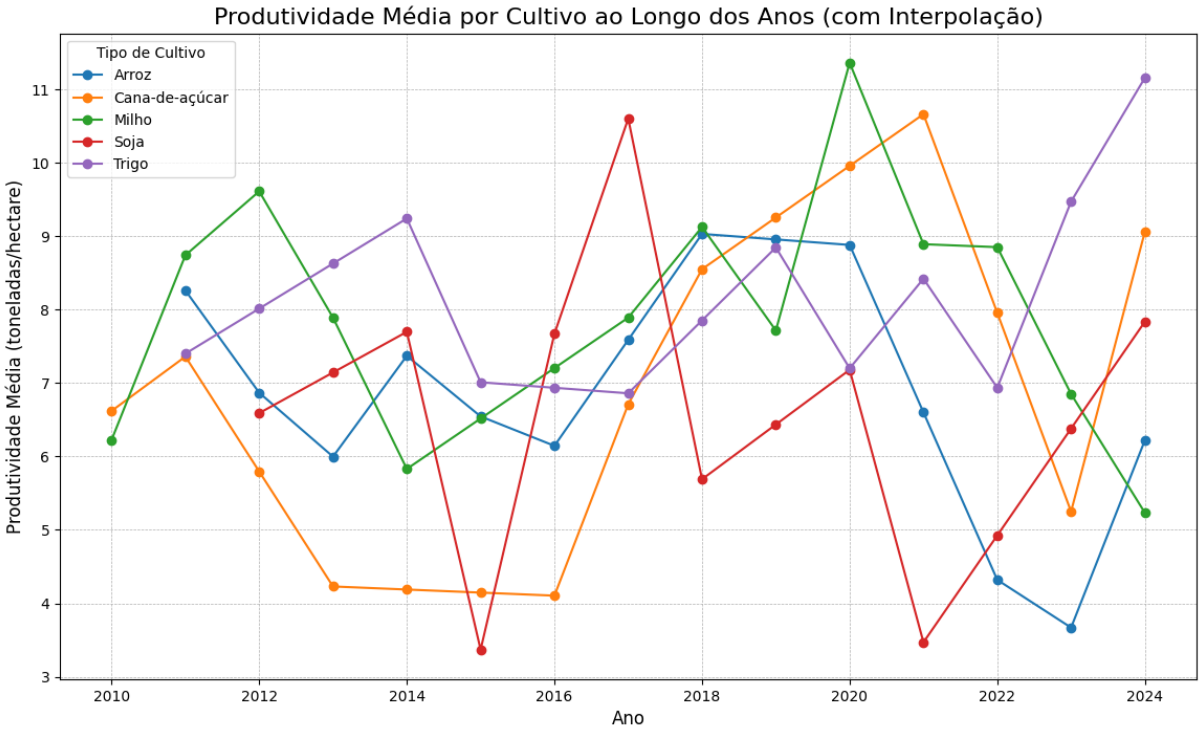

Figura 4.2: Séries temporais da produtividade ajustadas por interpolação

Interpretação Final: Agora com um gráfico contínuo, a análise se torna muito mais clara. Vemos que o **Trigo** e o **Milho**, apesar de voláteis, atingiram os maiores picos de produtividade. A **Soja** também mostra picos notáveis, mas com grande variação. Esta visualização é uma ferramenta estratégica poderosa, permitindo que a cooperativa identifique não apenas os cultivos mais produtivos em média, mas também os mais consistentes ao longo do tempo, auxiliando em decisões de investimento e planejamento de safra.

4.2 Experimento 2: Análise de Piscicultura

No nosso primeiro experimento, atuamos como analistas para uma grande cooperativa agrícola. Agora, vamos mudar de escala e aplicar nossas habilidades em um cenário igualmente vital para a economia amazônica, a pequena produção. O Senhor João é um piscicultor da região que cria três espécies de peixes — Tambaqui, Matrinxã e Jaraqui — e precisa da nossa ajuda para tomar uma decisão estratégica. Ele quer investir na espécie que lhe trará maior retorno financeiro no próximo ano. Este é um desafio clássico de análise de negócios, utilizar dados históricos para guiar decisões futuras e maximizar a rentabilidade.

4.2.1 Análise do Problema e dos Dados

O objetivo deste experimento é analisar os dados de produção e custo do Senhor João para identificar qual espécie de peixe foi a mais rentável ao longo do último ano. Nossa análise servirá como base para uma recomendação de investimento para o próximo ciclo de produção. O primeiro passo é carregar e analisar o conjunto de dados através do aquivo `dados_piscicultura.csv`, o qual contém os seguintes campos:

- **PEIXE:** O nome da espécie (Tambaqui, Matrinxã, Jaraqui).

- **PRODUCAO:** A quantidade de peixe produzida (em kg) em um determinado mês.
- **CUSTO_MEDIO:** O custo médio de produção por quilo para aquele peixe, naquele mês.
- **MES:** O mês em que o registro foi feito.

Caso Prático

Passo 1: Leitura e Inspeção Inicial dos Dados
Raciocínio: Como em todo projeto, nosso primeiro passo é carregar os dados do repositório online para um DataFrame do Pandas. Utilizaremos `pd.read_csv()` com a URL do arquivo. É importante notar que este arquivo utiliza ponto e vírgula como separador, então usaremos o parâmetro `sep=';'`. Em seguida, faremos a inspeção com `.head()` e `.info()` para verificar se os dados foram carregados corretamente.

Copie e Teste!

```
import pandas as pd

# URL para o arquivo CSV bruto no GitHub
url = 'https://raw.githubusercontent.com/CITHA-AM/Python/refs/heads/main/Modulo%204/dados_piscicultura.csv'

# Passo 1: Leitura e Inspeção Inicial dos Dados
# Para carregar os dados nesse arquivo, é preciso especificar o separador como ponto e vírgula
df_peixes = pd.read_csv(url, sep=';')

print("--- Dados de Piscicultura Carregados ---")
print(df_peixes.head())

print("\n--- Informações e Estrutura do DataFrame ---")
df_peixes.info()
```

Saída Esperada

```
--- Dados de Piscicultura Carregados ---
      PEIXE  PRODUÇÃO  CUSTO_MEDIO        MES
0  TAMBAQUI       100          5.0    JANEIRO
1  TAMBAQUI       100          6.0  FEVEREIRO
2  TAMBAQUI       120          6.1      MARÇO
3  TAMBAQUI       200          6.2      ABRIL
4  TAMBAQUI        90          6.3       MAIO

--- Informações e Estrutura do DataFrame ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36 entries, 0 to 35
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype
```

```
---  ------         --------------  -----
 0   PEIXE          36 non-null     object
 1   PRODUÇÃO       36 non-null     int64
 2   CUSTO_MEDIO    36 non-null     float64
 3   MES            36 non-null     object
dtypes: float64(1), int64(1), object(2)
memory usage: 1.3+ KB
```

4.2.2 Implementação da Solução em Python

Com os dados do Senhor João carregados e verificados, podemos começar a responder sua pergunta principal: qual peixe é o mais rentável? Para chegar a uma recomendação sólida, vamos dividir nossa análise em duas etapas: primeiro, calcularemos a produção total de cada espécie e, em segundo, criaremos uma métrica de rentabilidade que relaciona a produção com o custo total.

Caso Prático

Passo 2: Calcular a Produção Total e Média por Espécie
Raciocínio: Antes de analisar a rentabilidade, é importante entender o volume de produção de cada peixe. Vamos agrupar o DataFrame pela coluna `PEIXE` e, em seguida, aplicar múltiplas agregações (`.agg()`) na coluna `PRODUCAO` para calcular tanto a produção total (`sum`) quanto a média mensal (`mean`) para cada espécie.

Copie e Teste!

```python
# Passo 2: Análise de Produção por Peixe
print("--- 2. Análise de Produção por Peixe ---")

# Agrupando por peixe e calculando a soma e a média da produção
analise_producao = df_peixes.groupby('PEIXE')['PRODUÇÃO'].agg(['
    sum', 'mean'])
analise_producao = analise_producao.rename(columns={'sum': '
    Produção Total (kg)', 'mean': 'Produção Média Mensal (kg)'})

print(analise_producao.round(2))
```

Saída Esperada

```
--- 2. Análise de Produção por Peixe ---
          Produção Total (kg)  Produção Média Mensal (kg)
PEIXE
JARAQUI                   995                       82.92
MATRINXA                  420                       35.00
TAMBAQUI                 1420                      118.33
```

Interpretação: A análise de produção mostra que o **Tambaqui** é, de longe, o peixe com maior volume de produção, tanto total quanto em média mensal. O **Matrinxã**, por outro lado, tem a menor produção. Esta informação, no entanto, conta apenas metade da história. Um peixe pode ser muito produtivo, mas ter um custo de produção tão alto que sua rentabilidade seja baixa.

Caso Prático

Passo 3: Calcular a Rentabilidade de Cada Espécie
Raciocínio: Para encontrar o peixe mais rentável, precisamos de uma métrica que compare o que foi produzido com o que foi gasto. Primeiro, vamos calcular o custo total para cada peixe, que é a soma da `PRODUCAO` de cada mês multiplicada pelo `CUSTO_MEDIO` daquele mês. Em seguida, criaremos nosso indicador de rentabilidade, dividindo a `Produção Total (kg)` pelo `Custo Total (R$)`. O resultado nos dirá quantos quilos de peixe o Senhor João consegue produzir para cada R$1,00 investido.

Copie e Teste!

```
# Passo 3: Análise de Custo e Rentabilidade
print("\n--- 3. Análise de Rentabilidade ---")
# Custo Total = Soma(Produção de cada mês * Custo Médio de cada
    mês)
df_peixes['CUSTO_TOTAL_MES'] = df_peixes['PRODUÇÃO'] * df_peixes['
    CUSTO_MEDIO']

# Agrupando para obter os totais por peixe
custo_total = df_peixes.groupby('PEIXE')['CUSTO_TOTAL_MES'].sum()
producao_total = df_peixes.groupby('PEIXE')['PRODUÇÃO'].sum()

# Calculando a rentabilidade
rentabilidade = producao_total / custo_total

print("Rentabilidade (kg produzidos por R$ investido):")
print(rentabilidade.round(2))
print(f"\nO peixe mais rentável é o: {rentabilidade.idxmax()}")
```

Saída Esperada

```
--- 3. Análise de Rentabilidade ---
Rentabilidade (kg produzidos por R$ investido):
PEIXE
JARAQUI      0.20
MATRINXA     0.08
TAMBAQUI     0.15
Name: rentabilidade, dtype: float64

O peixe mais rentável é o: JARAQUI
```

Interpretação: Esta é a descoberta mais importante para o Senhor João. Embora o Tambaqui seja o mais produtivo, o **Jaraqui é o mais rentável**. Para cada real investido, o Jaraqui retorna 0.20 kg de peixe, enquanto o Matrinxã tem o pior retorno, com apenas 0.08 kg por real. Essa análise sugere que, para maximizar o lucro, o foco do investimento deveria ser no Jaraqui.

4.2.3 Apresentação dos Resultados

A análise de rentabilidade nos deu uma resposta clara sobre qual peixe oferece o melhor retorno financeiro. Agora, vamos criar uma visualização que não apenas complementa, mas aprofunda essa descoberta. Para uma recomendação de negócios completa, o Senhor João precisa entender duas histórias: a **operacional** (o volume de produção) e a **financeira** (o custo associado a essa produção). Vamos apresentar as duas lado a lado.

Caso Prático

Passo 4: Visualizar a Análise Operacional e Financeira
Raciocínio: A melhor maneira de comparar a produção com o custo é criar uma figura com dois gráficos alinhados (subplots). O gráfico de cima mostrará a produção mensal de cada peixe, nos dando a visão operacional. O gráfico de baixo mostrará o custo total mensal, nos dando a visão financeira. Ao alinhá-los pelo mesmo eixo de meses, podemos facilmente identificar como os picos de custo se relacionam com os picos de produção. Para isso, precisaremos pivotar nossos dados duas vezes: uma para os valores de `PRODUCAO` e outra para `CUSTO_TOTAL_MES`.

Copie e Teste!

```
import matplotlib.pyplot as plt

# Passo 4: Gerando visualização integrada (Produção vs. Custo)
# --- Preparando os dados para pivotar ---
df_prod_pivot = df_peixes.pivot(index='MES', columns='PEIXE',
    values='PRODUCAO')
df_custo_pivot = df_peixes.pivot(index='MES', columns='PEIXE',
    values='CUSTO_TOTAL_MES')

# Ordenando os meses
meses_ordem = ['JANEIRO', 'FEVEREIRO', 'MARCO', 'ABRIL', 'MAIO', '
    JUNHO', 'JULHO', 'AGOSTO', 'SETEMBRO', 'OUTUBRO', 'NOVEMBRO', '
    DEZEMBRO']
df_prod_pivot = df_prod_pivot.reindex(meses_ordem)
df_custo_pivot = df_custo_pivot.reindex(meses_ordem)

# --- Criando a figura com 2 subplots ---
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(12, 10),
    sharex=True)
fig.suptitle('Análise Operacional e Financeira da Piscicultura',
    fontsize=18, fontweight='bold')
```

```
# --- Plot 1: Produção Mensal (kg) ---
df_prod_pivot.plot(kind='line', marker='o', ax=axes[0])
axes[0].set_title('Ciclo de Produção Mensal', fontsize=14)
axes[0].set_ylabel('Produção (kg)')
axes[0].legend(title='Peixe')
axes[0].grid(True, linestyle='--', linewidth=0.5)

# --- Plot 2: Custo Total Mensal (R$) ---
df_custo_pivot.plot(kind='line', marker='o', ax=axes[1])
axes[1].set_title('Custo Total de Produção Mensal', fontsize=14)
axes[1].set_ylabel('Custo Total (R$)')
axes[1].set_xlabel('Mês')
axes[1].legend(title='Peixe')
axes[1].grid(True, linestyle='--', linewidth=0.5)

# Ajustes finais
plt.xticks(ticks=range(len(meses_ordem)), labels=meses_ordem,
   rotation=45)
plt.tight_layout(rect=[0, 0, 1, 0.95]) # Ajusta o super-título
plt.show()
```

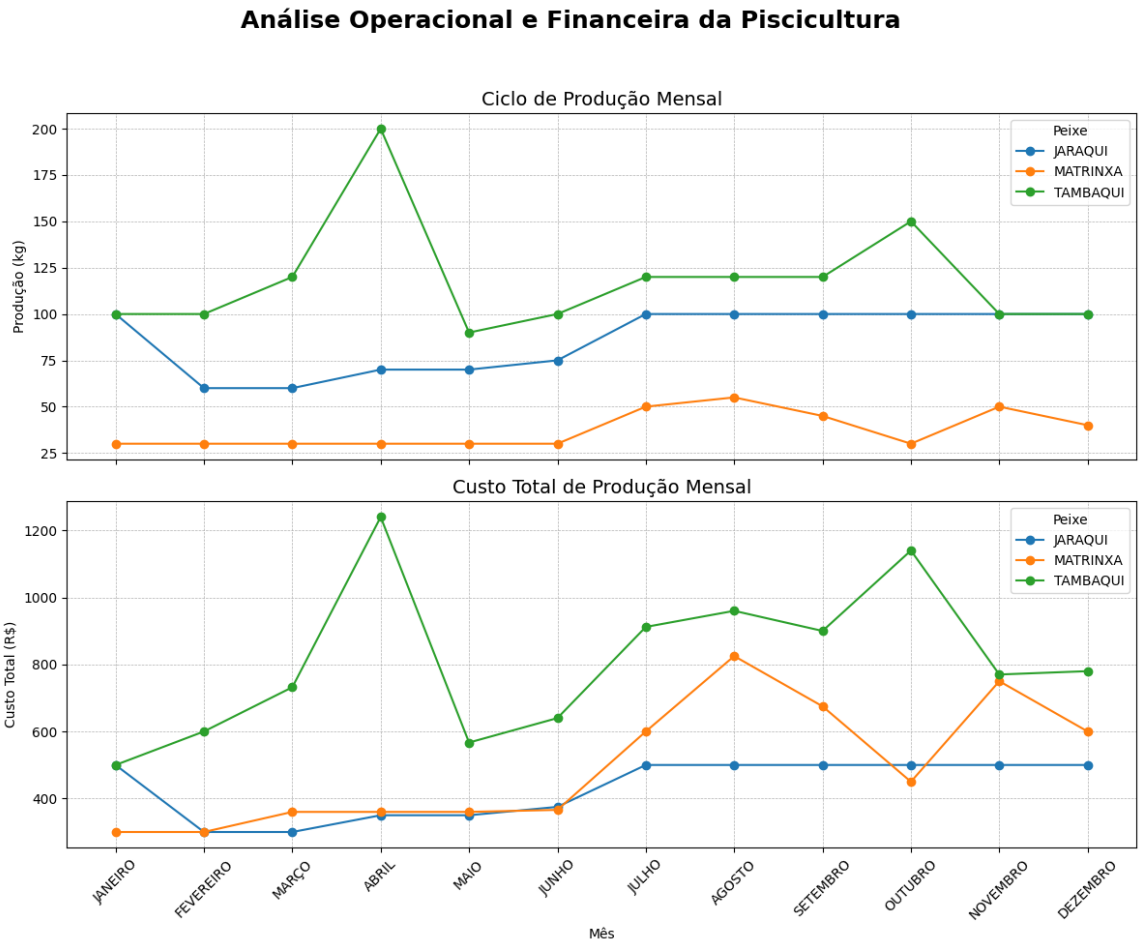

Figura 4.3: Séries temporais da produtividade e custo total de cada espécie

Interpretação Final e Recomendação para o Senhor João: A visualização combinada nos oferece um panorama completo do negócio. Analisando o primeiro gráfico a figura 4.3, há a confirmação de que o **Tambaqui** tem o maior pico de produção em Abril. No entanto, o gráfico de baixo revela a informação crucial: esse pico de produção vem acompanhado de um pico de **custo** igualmente expressivo, superando R$1200,00 naquele mês.

Por outro lado, o **Jaraqui**, apesar de ter uma produção menor, apresenta um custo de produção muito mais baixo e estável ao longo do ano. O **Matrinxã** se mostra inviável, com baixa produção e custos relativamente altos para o volume que gera.

Recomendação Final: A análise visual reforça a conclusão da nossa análise de rentabilidade. Para maximizar os lucros e ter uma operação financeira mais previsível, a recomendação para o Senhor João é focar o investimento no **Jaraqui**. O Tambaqui, embora produtivo, exige um capital de giro muito maior e oferece uma margem de retorno menor. Esta análise integrada (operacional + financeira) fornece uma base de dados sólida para uma decisão de negócios estratégica e segura.

Considerações Finais: Sua Jornada como Solucionador de Problemas

Conhecendo um pouco mais!

Materiais Práticos do Módulo 4
Lembre-se que todos os scripts, datasets e o notebook para este módulo estão disponíveis em nosso repositório no GitHub. Para Instruções detalhadas sobre como utilizar os materiais, consulte a subseção 1.5 Considerações do Módulo 1

- **Scripts e Datasets do Módulo 4:**
  `https://github.com/CITHA-AM/Python/tree/main/Modulo%204`
- **Notebook do Módulo 4 (Colab):**
  `https://colab.research.google.com/github/CITHA-AM/Python/blob/main/Modulo%204.ipynb`

Parabéns por chegar ao final desta jornada! Ao concluir os experimentos práticos do Módulo 4, você completou um ciclo de aprendizado que vai muito além de simplesmente escrever código. Você aprendeu a pensar e a agir como um verdadeiro solucionador de problemas da era digital.

Vamos recapitular o caminho que percorremos juntos:

- No **Módulo 1**, você foi alfabetizado na linguagem Python, aprendendo sua sintaxe, suas estruturas de dados e a lógica fundamental que governa a programação.
- No **Módulo 2**, você deu um passo em direção à organização e à robustez, aprendendo a modularizar seu código com funções e a dar ”memória”aos seus programas através da manipulação de arquivos.
- No **Módulo 3**, você mergulhou no universo da análise de dados, dominando as bibliotecas Pandas e Matplotlib para transformar dados brutos em tabelas organizadas e, por fim, em histórias visuais claras e impactantes.

- Finalmente, no **Módulo 4**, você integrou todas essas habilidades para resolver desafios práticos e relevantes para o contexto amazônico, analisando dados agrícolas e de piscicultura para extrair insights e guiar decisões estratégicas.

A principal lição deste livro não está nos comandos que você decorou, mas na metodologia que você aprendeu a aplicar: entender um problema, carregar e limpar os dados, realizar análises e, o mais importante, comunicar suas descobertas.

Os conhecimentos que você adquiriu aqui são a base para inúmeras inovações. Conforme a missão do projeto CITHA, esperamos que você utilize essa nova habilidade não apenas como uma ferramenta técnica, mas como um meio para criar soluções sustentáveis, otimizar recursos e gerar um impacto positivo na bioeconomia e na qualidade de vida na Amazônia.

O aprendizado é um processo contínuo. Agora que você tem esta base sólida, o convidamos a explorar novos territórios: aprofunde-se em bibliotecas de Machine Learning como Scikit-learn, explore a criação de painéis interativos com Streamlit ou Dash, ou automatize tarefas do seu dia a dia.

A sua expedição pelo mundo Python está apenas começando. Continue curioso, continue praticando e continue usando a tecnologia para construir um futuro melhor.

Referências Bibliográficas

VAN ROSSUM, Guido. Python: a general-purpose programming language. In: **Python Software Foundation**, 2024. Disponível em: `https://www.python.org/`. Acesso em: 5 jul. 2025.

LUTZ, Mark. **Learning Python**. 5. ed. Sebastopol, CA: O'Reilly Media, 2013.

ALMEIDA, Marcus. Pandas Python: o que é, para que serve e como instalar. In: **Alura**, 2023. Disponível em: `https://www.alura.com.br/artigos/pandas-o-que-e-para-que-serve-como-instalar`. Acesso em: 19 nov. 2024.

MATTHES, Eric. **Curso Intensivo de Python**: uma Introdução Prática e Baseada em Projetos à Programação. São Paulo: Novatec Editora, 2023.

MENEZES, Nilo Ney Coutinho. **Introdução à programação com Python**: algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

MUELLER, John Paul. **Começando a Programar em Python Para Leigos**. 2. ed. Rio de Janeiro: Alta Books, 2020.

CRUZ, Felipe. **Python**: escreva seus primeiros programas. São Paulo: Casa do Código, 2015.

MCKINNEY, Wes. **Python para Análise de Dados**: tratamento de dados com Pandas, NumPy e IPython. São Paulo: Novatec, 2018.

PAIVA, Fábio Augusto Procópio de; NASCIMENTO, João Maria Araújo do; MARTINS, Rodrigo Siqueira; SOUZA, Givanildo Rocha de. **Introdução a Python com aplicações de sistemas operacionais**. Natal: [s.n.], 2020.

RAMALHO, Luciano. **Python Fluente**: programação clara, concisa e eficaz. São Paulo: Novatec Editora, 2015.

VASILIEV, Yuli. **Python Para Ciência de Dados**: uma Introdução Prática. São Paulo: Novatec Editora, 2023.

FOTO DE SIMON PETEREIT

ISBN: 978-65-01-76116-9

CITHA

Capacitação e Interiorização em
Tecnologias Habilitadoras na Amazônia