Pós-Graduação em Inteligência Artificial Generativa e LLMs

PUC-Rio

Desmistificando a Programação de IA

Notas de Aula

Vítor Wilher

DPIA · 2025.2

Versão 1.0

Resumo. Fundamentos de Python aplicados a IA: sintaxe, estruturas de dados, funções, bibliotecas, NumPy, Pandas e análise de dados.

Palavras-chave: Python; NumPy; Pandas; análise de dados

Notas de estudo elaboradas a partir do material das aulas.

Índice

1 Visão Geral da Disciplina

A disciplina **DPIA — Desmistificando a Programação de IA** cumpre um papel de fundação dentro do programa de IA Generativa e LLMs. Seu propósito é retirar da programação de Inteligência Artificial o véu de complexidade que costuma afastar profissionais vindos de outras áreas, mostrando que por trás dos modelos mais sofisticados existe um conjunto relativamente pequeno de ideias: estruturas de dados bem escolhidas, operações vetorizadas, funções reutilizáveis e um método disciplinado de olhar para os dados. As aulas são conduzidas pela Prof. Manoela Kohler, do ICA/PUC-Rio.

O percurso começa longe do teclado. A primeira aula estabelece o vocabulário conceitual — o que é Inteligência Artificial, o que é **Machine Learning**, o que é **Deep Learning**, como esses campos se encaixam uns dentro dos outros — e situa historicamente a área, com seus ciclos de euforia e de "invernos". Essa contextualização importa porque define o que se espera de um projeto de IA e quais equívocos costumam fazer projetos fracassarem antes mesmo da primeira linha de código.

A partir daí, o curso mergulha em Python. Um bloco extenso (aulas 2, 3 e 4) percorre os fundamentos da linguagem: comentários, indentação, variáveis, tipos de dados, operadores, conversão de tipos, interação com o usuário, estruturas condicionais e estruturas de repetição. A aula 5 consolida o tema de funções e importação de bibliotecas, e a aula 6 introduz o **NumPy**, dando o salto conceitual das listas nativas para arrays vetorizados. As aulas 7 e 8 estendem esse ferramental para **DataFrames**, manipulação de dados e noções de estatística; a aula 9 aplica tudo em estudos de caso reais.

A disciplina se encerra na aula 10 com **LLM para Programação**, fechando um arco elegante: depois de aprender a programar com as próprias mãos, o aluno aprende a programar em parceria com modelos de linguagem — copilotos de código, padrões de prompt eficazes e, sobretudo, os cuidados críticos que separam o uso produtivo do uso ingênuo dessas ferramentas.

2 Introdução a IA

Esta aula tem por objetivo declarado tornar o aluno capaz de definir Inteligência Artificial e suas subáreas, explicar como o Deep Learning ajuda a resolver limitações clássicas do Machine Learning, compreender os desenvolvimentos históricos e os ciclos de inverno e hype da IA, diferenciar a IA moderna da IA clássica, entender possibilidades de aplicação, conhecer o fluxo de um projeto de ML e reconhecer equívocos comuns na área. Parte das notas provém do curso *Artificial Intelligence 501*, formulado pela Intel.

2.1 IA como infraestrutura

O ponto de partida é a analogia de Andrew Ng, da Universidade de Stanford: "Há cerca de 100 anos, a eletricidade transformou toda indústria importante. A IA avançou ao ponto em que tem o poder de transformar todo setor relevante nos próximos anos." A comparação não é retórica vazia. Ela sugere que a IA deixou de ser um produto vertical, específico de um nicho, e passou a ser uma camada horizontal de infraestrutura — algo que atravessa transporte, mídias sociais, saúde, finanças e o cotidiano.

2.2 Definições encaixadas: IA, ML e DL

Inteligência Artificial é o campo mais amplo. Dentro dele está o **Aprendizado de Máquina (Machine Learning)**, definido de forma precisa nos slides: são algoritmos que *aprendem ao ver repetidamente um conjunto de dados, ao invés de serem explicitamente programados por seres humanos*. Essa é a distinção crucial em relação à programação tradicional. No paradigma clássico, o programador escreve as regras; no aprendizado de máquina, o programador escreve o procedimento pelo qual as regras são descobertas a partir dos dados.

A terminologia básica de ML é introduzida a partir de um exemplo canônico: uma base para classificação de espécies de flores, onde cada linha é uma observação e cada coluna uma característica.

O exemplo de aplicação usado é a **detecção de transações fraudulentas de cartão de crédito**. O trabalho do analista é identificar características (features) para que o algoritmo aprenda que combinações delas sugerem atividade incomum:

- Hora da transação
- Valor da transação
- Localização
- Categoria da compra

Esse exercício mostra o ML clássico funcionando bem: as características são conhecidas, tabuláveis e fazem sentido para um especialista de domínio.

2.3 A limitação que motiva o Deep Learning

Em seguida vem a pergunta que quebra o paradigma anterior: suponha que você queira determinar se uma imagem é de um gato ou de um cão — *que características você usaria*? Não há uma lista tabular óbvia. Não existe uma coluna "orelha pontuda" pré-computada nos pixels. É exatamente aí que entra o **Deep Learning**.

A **Aprendizagem Profunda** resolve o problema aprendendo as próprias representações, organizadas em **diferentes níveis de abstração**: camadas iniciais capturam bordas e texturas, camadas intermediárias capturam partes de objetos, camadas finais capturam conceitos. O ser humano não precisa mais projetar as características à mão; a rede as descobre.

2.4 De NLP a modelos multimodais

Em processamento de linguagem natural, os slides marcam a trajetória das **LSTM** para os **Transformers**. A motivação para modelos **multimodais** é apresentada em três objetivos:

- Aumentar contexto e robustez
- Modelar tarefas complexas, como responder perguntas sobre imagens
- Melhorar o entendimento do mundo real

O argumento é intuitivo e memorável: *nós somos multimodais* — percebemos o mundo simultaneamente por visão, audição e linguagem, então faz sentido que os modelos também o façam.

2.5 Foundation Models e agentes

Os **Foundation Models** são apresentados como a nova unidade de construção da IA: modelos grandes, treinados em larga escala, reaproveitáveis para muitas tarefas. O exemplo concreto em visão computacional é o **Segment Anything Model (SAM)**, com site oficial e demo em `segment-anything.com`, artigo científico em `arxiv.org/abs/2304.02643` e código aberto no GitHub sob licença Apache 2.0 (uso comercial liberado com os devidos créditos). Uma variação citada é o **Grounded SAM**.

Sobre **agentes de IA**, a aula descreve o ciclo agente-sistema em cinco etapas:

1. Receber instrução do usuário
2. Raciocinar: definir estratégia
3. Planejar: escolher ferramentas e sequência
4. Executar: invocar ferramentas
5. Observar e ajustar respostas e ferramentas

Esse ciclo é a espinha dorsal conceitual de praticamente todo sistema agêntico moderno.

2.6 História: hypes e invernos

A linha do tempo apresentada organiza a área em ciclos alternados de entusiasmo e retração.

Early AI (a partir de 1950). Em 1950, Alan Turing desenvolve o teste de Turing para avaliar a capacidade de máquinas exibirem comportamento inteligente. Em 1956, a IA é aceita como campo na Conferência de Dartmouth. Em 1957, Frank Rosenblatt inventa o perceptron, precursor das redes neurais modernas. Em 1959, Arthur Samuel publica um algoritmo para um programa de damas usando aprendizado de máquina.

O primeiro inverno da IA. Em 1966, o comitê da ALPAC (chefiado por John R. Pierce) avalia técnicas de IA para tradução automática e conclui que o retorno não compensou o investimento. Em 1969, Marvin Minsky publica um livro sobre as limitações do perceptron, retardando a pesquisa em redes neurais. Em 1973, o relatório Lighthill destaca o fracasso da IA. Os dois relatórios levam a cortes no financiamento governamental e ao primeiro "A.I. Winter".

O boom dos anos 1980. Surgem os **sistemas especialistas**, com regras programadas para imitar especialistas humanos, rodando em mainframes com linguagens como LISP. Foram as primeiras tecnologias de IA amplamente utilizadas: no auge, dois terços das empresas da Fortune 500 os utilizavam. Em 1986, o algoritmo de **backpropagation** torna-se capaz de treinar perceptrons de múltiplas camadas, renovando o interesse na área.

Outro inverno (final dos 1980 e início dos 1990). O progresso dos sistemas especialistas desacelera; eles passam a ser absorvidos por aplicativos corporativos gerais como SAP e Oracle, que rodavam em PCs em vez de mainframes. As redes neurais não escalam para problemas maiores e mais complexos, e o interesse empresarial diminui.

Aprendizado de máquina clássico (final dos 1990 e início dos 2000). Avanços no algoritmo **SVM** o tornam o método de escolha. Soluções de IA obtêm sucesso em reconhecimento de voz, diagnóstico médico e robótica. O Deep Blue derrota Garry Kasparov no xadrez, e o motor de busca do Google é lançado usando inteligência artificial.

A ascensão do Deep Learning (a partir de 2006). Geoffrey Hinton publica um artigo sobre pré-treinamento não supervisionado que permite treinar redes mais profundas. Em 2009, a base **ImageNet** é apresentada na conferência CVPR; em 2010 ocorre a primeira competição ImageNet.

IA moderna (2012 até o presente). Em 2012, o Deep Learning bate os benchmarks anteriores no ImageNet. Em 2013, é usado para entender o significado conceitual de palavras. Em 2014, avanços semelhantes aparecem em tradução — com ganhos em pesquisa na web, pesquisa em documentos e sumarização — e algoritmos de visão passam a descrever fotos. Em 2015 surge o **TensorFlow**. Em 2016, o AlphaGo da DeepMind, desenvolvido por Aja Huang, derrota Lee Se-dol, o maior campeão de Go. Em março de 2023 é fundada a xAI, por Elon Musk.

Para 2024 e 2025, os slides listam como marcos: **LLM**, **SLM**, **VLM**, modelos multimodais, Foundation Models, agentes de IA, grande evolução de modelos proprietários, DeepSeek, open-source, **MoE**, ética e regulamentação.

2.7 Aplicações e fluxo de trabalho

As aplicações destacadas cobrem transporte, mídias sociais e o dia a dia. Sobre o **fluxo de trabalho de projetos de ML em geral**, a aula apresenta um fluxograma de referência que vai da definição do problema à entrega em produção, passando por dados, modelagem e avaliação.

2.8 Seis equívocos comuns

Esta é, talvez, a parte mais valiosa da aula do ponto de vista profissional. São seis mal-entendidos frequentes:

Equívoco 1 — o unicórnio. Cientistas de dados especialistas em todas as áreas são chamados de unicórnio, e são raros justamente por isso. Equipes bem-sucedidas contêm pessoas de diversas formações e habilidades: alguns se distinguem em comunicação, outros em estatística. Boas equipes têm especialistas em três áreas principais: **negócios**, **ciência** e **engenharia**.

Equívoco 2 — foco exclusivo em pesquisa e algoritmos. Equipes de ciência de dados não podem focar apenas em pesquisa. Boas equipes conseguem identificar problemas, comunicar suas descobertas e trabalhar com a engenharia para achar o melhor método de produção.

Equívoco 3 — sistemas complexos são melhores. A solução mais complicada nem sempre é a melhor. Equipes tendem a ser mais bem-sucedidas quando começam do básico e depois passam para técnicas mais complexas. Modelos complexos podem ser mais precisos, mas são menos interpretáveis, mais propensos a falhar de forma imprevisível e mais difíceis de sustentar. Começar do básico também garante alinhamento com as necessidades do negócio.

Equívoco 4 — as técnicas são iguais em qualquer indústria. As técnicas para limpar, armazenar e modelar dados são de fato similares entre setores, mas é necessário **conhecimento de domínio** para entender quais são os dados e os problemas mais relevantes.

Equívoco 5 — projetos começam bem definidos. Projetos de ciência de dados normalmente são exploratórios e experimentais. Geralmente não é claro quão difícil será a solução até que se explore os dados. Gerentes de produto devem trabalhar com a equipe e demais envolvidos para gerenciar expectativas.

Equívoco 6 — os melhores modelos são os mais avançados. Há mais desafios na escolha de um modelo do que sua capacidade preditiva. Alguns modelos podem ser lentos ou complicados demais para produção; outros podem não ser interpretáveis, tornando investidores mais relutantes.

3 Introdução ao Python

3.1 Lógica, algoritmos e programação

Antes de qualquer sintaxe, os slides posicionam a programação dentro da lógica. A lógica é a parte da filosofia que trata das formas do pensamento em geral e das operações intelectuais que visam determinar o que é verdadeiro ou não. No universo da tecnologia da informação, a lógica é a **organização e o planejamento das instruções e assertivas em um algoritmo**, a fim de viabilizar a implantação de um programa. Daí a definição sintética: *programar nada mais é do que a organização coesa de uma sequência de instruções voltadas à resolução de um problema de forma lógica.*

3.2 Por que Python

A filosofia da linguagem é resumida em sete pontos: totalmente gratuito; usabilidade; orientado a objetos; poderoso; ferramentas gráficas poderosas; flexível; e vasta comunidade. As características técnicas reforçam: open-source, compatibilidade, uso mundial pela academia e pela indústria, e atualização constante com novas bibliotecas de uso livre.

Entre os ambientes de desenvolvimento citados estão **Jupyter**, **Spyder**, **PyCharm**, **VS Code** e o **Google Colab**. A conclusão sobre qual escolher é deliberadamente pragmática: *a melhor IDE é aquela que você se sente mais à vontade ao utilizar.*

3.3 Comentários e indentação

Quando o programa cresce, torna-se difícil de ler e manter — por isso é boa prática inserir documentação e notas, chamadas **comentários**. O comentário em Python começa com o sinal de hash (**#**) e continua até o fim da linha; o interpretador o ignora. Os slides distinguem comentário em linha, comentário em bloco e comentário de documentação para funções, módulos, pacotes e classes.

Python usa **indentação** para delimitar blocos, em vez de chaves. Tabs e espaços são suportados.

```
# Comentário em linha

def area_circulo(raio):
    """Docstring: documenta o que a função faz."""
    return 3.14159 * raio ** 2
```

3.4 Variáveis e tipos de dados

Tipos numéricos. O tipo numérico representa dados com valor numérico, podendo ser inteiro, número flutuante ou complexo — as classes `int`, `float` e `complex`. Os inteiros contêm números positivos ou negativos sem fração ou decimal. O float é um número real com representação de ponto flutuante, especificado por um ponto decimal. O número complexo é especificado como parte real mais parte imaginária seguida de `j`.

Booleano. Tipo com um de dois valores internos, Verdadeiro ou Falso, indicado pela classe `bool`.

Sequências. Uma sequência é a coleção ordenada de tipos de dados semelhantes ou diferentes, permitindo armazenar vários valores de forma organizada e eficiente. Existem três tipos de sequência em Python:

- **String** (`str`): matrizes de bytes que representam caracteres; uma coleção de um ou mais caracteres entre aspas simples, duplas ou triplas. É possível acessar elementos dentro de uma string por índice.
- **Lista** (`list`): semelhante a matrizes de outras linguagens, mas **não precisa ser homogênea** — uma única lista pode conter inteiros, strings e objetos. Listas são **mutáveis**, ordenadas e têm contagem definida.
- **Tupla** (`tuple`): coleção ordenada de objetos Python, semelhante a uma lista, com valores de qualquer tipo indexados por inteiros. A diferença importante é que tuplas são **imutáveis**.

Set. Coleção **não ordenada**, iterável, mutável e **sem elementos duplicados**. A ordem dos elementos é indefinida. A principal vantagem sobre a lista é possuir um método altamente otimizado para verificar se um elemento específico está contido no conjunto.

Dicionário. Coleção não ordenada de valores de dados usada para armazenar dados como um mapa. Diferentemente de outros tipos que mantêm apenas um valor único como elemento, o dicionário mantém um par **chave:valor**. Chave e valor são separados por dois pontos; cada par é separado por vírgula.

```
inteiro = 6
flutuante = 3.5
verdade = True
texto = "Python"

lista = [1, "dois", 3.0]
tupla = (1, 2, 3)          # imutável
conjunto = {1, 2, 3}       # sem duplicatas
dicionario = {"nome": "The Shining", "ano": 1980}
```

3.5 Operadores, type casting e entrada do usuário

A aula cobre os **operadores** (aritméticos, de comparação, lógicos), o **type casting** — conversão explícita entre tipos, como `int()`, `float()` e `str()` — e a **interação com o usuário** via `input()`.

3.6 Exercícios do primeiro bloco

Os exercícios propostos são cuidadosamente escolhidos para expor sutilezas da linguagem:

1. Criar uma variável com valor 6 (inteiro) e outra com valor 2 (inteiro), dividir a primeira pela segunda e observar o tipo do resultado — o objetivo é entender por que a divisão retorna `float` mesmo entre inteiros.
2. Dois amigos dividem o lucro de um site de consultoria em projetos de IA. O lucro mensal é de 8k; o amigo1 tem direito a 30% e o restante pertence ao amigo2. Calcular o lucro total do ano e o lucro de cada um.
3. Prever mentalmente o resultado de `5 + 3 * 10 / 3 == 15` e confirmar no Python — um exercício sobre precedência de operadores e ponto flutuante.

O segundo conjunto usa dados de um filme:

```
movieName = "The Shining"
Actors = ["Jack Nicholson", "Shelley Duvall", "Danny Lloyd",
          "Scatman Crothers", "Barry Nelson"]
scores = [4.5, 4.0, 5.0]
comments = ["Best Horror Film I Have Ever Seen",
            "A truly brilliant and scary film from Stanley Kubrick",
            "A masterpiece of psychological horror"]
```

As tarefas: criar uma lista com os quatro elementos; imprimir a string com o nome do primeiro ator; imprimir a melhor avaliação do filme (score e comentário), usando as dicas `max(lista)` para o valor máximo e `lista.index(valor)` para recuperar o índice; e refazer o exercício salvando as variáveis em um **dicionário** em vez de uma lista. A comparação entre as duas versões é o ponto pedagógico: o dicionário nomeia os campos e dispensa a memorização de posições.

Há ainda o exercício de `input()`: perguntar quantos anos o usuário tem, imprimir a idade, seu tipo e o ano de nascimento (considerando que a pessoa já fez aniversário). A exigência de que o script funcione em qualquer ano em que venha a ser rodado força o uso do pacote `datetime`:

```
from datetime import datetime
datetime.now()        # data e hora corrente
datetime.now().year   # ano corrente
```

4 Introdução ao Python - Continuação

Esta aula dá continuidade ao mesmo material (`Aula01_02_03_Intro.pdf`), avançando das estruturas de dados para as **estruturas de controle**, com exercícios extras de apoio.

4.1 Estruturas condicionais

As estruturas condicionais permitem que o programa siga caminhos diferentes conforme uma expressão booleana. Em Python, são expressas por `if`, `elif` e `else`, com blocos delimitados por indentação.

```python
idade = 20

if idade < 12:
    print("Criança")
elif idade < 18:
    print("Adolescente")
else:
    print("Adulto")
```

4.2 Estruturas de repetição

Os slides apresentam lado a lado os dois laços fundamentais: **while loop** e **for loop**. O `for` percorre um iterável de tamanho conhecido; o `while` repete enquanto uma condição permanecer verdadeira.

```python
# for
for i in range(25):
    print(i)

# while
i = 0
while i < 25:
    print(i)
    i += 1
```

4.3 Exercícios de repetição

Os exercícios são construídos para que cada um seja resolvido **de duas formas**: com estrutura de repetição e sem ela — em geral usando NumPy. Essa duplicidade prepara o terreno para a aula de NumPy, onde a vetorização será justificada por desempenho.

1. Imprimir uma sequência de 25 números consecutivos, com e sem estrutura de repetição.
2. Imprimir a sequência acima duas vezes, usando `for` dentro de `for` ou `while` dentro de `while`.
3. Na lista `[12, 23, 11, 34, 13, 56, 102, 101, 13]`, imprimir somente os números pares. As dicas são o operador de resto `%` (por exemplo, `10 % 2` resulta em 0) e o operador de comparação `==`.
4. **Mega Sena**: sortear 6 números entre 1 e 60 em um sorteio, com e sem estrutura de repetição, usando `np.random.randint()`. A pergunta provocativa do slide é essencial: *estrutura de repetição sem nenhum tratamento e* ***`np.random.randint`*** *sem nenhum tratamento podem gerar algum problema?* A resposta é sim — números repetidos, o que é inválido em um sorteio. A dica para resolver corretamente é `np.random.choice()`, que permite amostragem sem reposição.
5. Simular o resultado de um dado de 6 faces jogado 7 vezes, com e sem estrutura de repetição.

O exemplo de uso fornecido é `np.random.randint(1, 61, 1)`, que sorteia 1 número entre 1 e 60.

```python
import numpy as np

# Com repeticao (pode repetir numeros)
sorteio = [np.random.randint(1, 61) for _ in range(6)]
```

```
# Sem repeticao (correto para Mega Sena)
sorteio = np.random.choice(range(1, 61), size=6, replace=False)
```

6. Sabendo que o fatorial é definido como `5! = 5 x 4 x 3 x 2 x 1 = 120`, criar um script que calcule o fatorial de um número qualquer, com e sem estrutura de repetição (neste último caso usando o pacote `math`).

4.4 Lei dos Grandes Números

O exercício mais conceitual do bloco é a verificação empírica da **Lei dos Grandes Números**, expressa nos slides como a convergência da média amostral para o valor esperado quando o número de experimentos cresce: $X_n \to E(X)$ quando $n \to \infty$.

A ilustração inicial usa um experimento binário, mostrando como a proporção observada se aproxima de 50% e 50% à medida que n cresce:

- 10 experimentos: 7/3, ou seja, 70% e 30%
- 100 experimentos: 52/48, ou seja, 52% e 48%
- 1000 experimentos: 502/498, ou seja, 50,2% e 49,8%

O exercício formal pede: testar a lei dos grandes números para N números aleatórios gerados com **distribuição normal** de média 0 e desvio padrão 1; criar um script que conte quantos desses números caem entre -1 e 1 e divida por N (o número de experimentos); sabendo que $E(X) = 68{,}2\%$, verificar que a média tende a esse valor conforme N aumenta.

```
import numpy as np

N = 1000
amostra = np.random.normal(0, 1, N)
proporcao = np.sum((amostra > -1) & (amostra < 1)) / N
print(proporcao)   # tende a 0.682 conforme N cresce
```

O valor de 68,2% não é arbitrário: é a fração da distribuição normal padrão contida em um desvio padrão em torno da média. O exercício conecta, portanto, programação, simulação e estatística em uma única atividade.

5 Introdução à Python - Parte 3

A terceira parte do bloco introdutório encerra o material `Aula01_02_03_Intro.pdf` consolidando o tema de **funções** e a prática dos exercícios extras. É a transição natural entre escrever scripts lineares e escrever código reutilizável.

5.1 Anatomia de uma função

Os slides decompõem uma função em três elementos: os **parâmetros de entrada**, o corpo da **função** propriamente dita e o **retorno**. Uma regra operacional é explicitada: *a função deve ser declarada antes de sua chamada.*

5.2 Por que funções

A justificativa é direta: as funções são o **primeiro passo para a reutilização de código**. Elas permitem definir um bloco de código reutilizável que pode ser usado repetidamente em um programa. Python já fornece várias funções internas, como `print()` e `len()`, mas o programador também pode definir as suas próprias.

```
def area_circulo(raio):
    import math
    return math.pi * raio ** 2

print(area_circulo(5))
```

5.3 Exercícios de funções

1. Criar uma função que calcule o fatorial de um número qualquer.
2. Usar a função `factorial` do pacote `math` para calcular o fatorial — comparando implementação própria com biblioteca padrão.
3. Criar uma função que receba o raio de um círculo e retorne sua área, utilizando o pacote `math`. As dicas são `math.pi` e, para potência, o operador `**` ou `math.pow`.
4. Alterar a função anterior para que retorne, além da área, também o **diâmetro** e o **perímetro**; colocar um **valor padrão** para o raio (5) e usar um **dicionário** para retornar os valores.

```
import math

def circulo(raio=5):
    return {
        "area": math.pi * raio ** 2,
        "diametro": 2 * raio,
        "perimetro": 2 * math.pi * raio,
    }
```

Este quarto exercício é particularmente rico: introduz simultaneamente **argumentos com valor padrão** e o retorno de múltiplos valores nomeados por meio de um dicionário — um padrão de projeto que evita a ambiguidade de retornar tuplas posicionais.

5.4 Encapsulando a Lei dos Grandes Números

O exercício de fechamento pede a criação de uma função para validar a lei dos grandes números. A função deve receber como parâmetro o número de experimentos e calcular a média dos experimentos no intervalo fixo de -1 a 1, sabendo que $E(X) = 68{,}2\%$, testando para n = 10, 1000, 1000000 e assim por diante.

Após a validação, o enunciado pede um refinamento: incluir um ou dois **parâmetros opcionais** para o intervalo a ser validado, com valores default de -1 e 1, e então testar outros intervalos verificando a lei.

```
import numpy as np
```

```python
def lgn(n, inferior=-1, superior=1):
    amostra = np.random.normal(0, 1, n)
    return np.mean((amostra > inferior) & (amostra < superior))

for n in [10, 1000, 1000000]:
    print(n, lgn(n))
```

A progressão pedagógica aqui é exemplar: o mesmo problema aparece primeiro como script, depois como função rígida e finalmente como função parametrizada e generalizável.

6 Declaração de funções e importação de bibliotecas

Material de slides não disponível para esta aula.

Esta aula é dedicada à consolidação prática de dois temas já introduzidos no bloco anterior: a **declaração de funções** e a **importação de bibliotecas**. O material de apoio consiste em notebooks (`Funcoes_e_Bibliotecas.ipynb` e as versões feitas em aula), o que indica um formato de laboratório, com o conteúdo desenvolvido ao vivo em código.

Do que o material de slides das aulas anteriores permite afirmar, o tema de bibliotecas aparece explicitamente no programa da disciplina como "Bibliotecas: instalação e utilização", e vários exercícios já exigiam importações concretas: `datetime` para obter o ano corrente, `math` para `factorial`, `pi` e `pow`, `numpy` para geração de números aleatórios e, mais adiante, `time` para medição de desempenho.

```python
import numpy as np              # importa com apelido
import math                     # importa o modulo inteiro
from datetime import datetime   # importa um objeto do modulo
```

As três formas acima cobrem os padrões usados ao longo de toda a disciplina: o **apelido** (`as np`) reduz a digitação em bibliotecas de uso intensivo, a importação do módulo inteiro preserva o namespace e evita colisões de nomes, e a importação seletiva traz apenas o que é necessário.

Quanto às funções, os elementos já estabelecidos e retomados aqui são a declaração com `def`, os parâmetros de entrada, os parâmetros com valor padrão, o retorno de valores (inclusive múltiplos, via dicionário) e a regra de que a função precisa estar declarada antes de ser chamada.

7 Numpy na Prática

7.1 O que é e por que existe

O **NumPy** é descrito nos slides como o pacote básico fundamental da linguagem Python para computação científica. Ele provê muitas das estruturas básicas, fornecendo ferramentas para integração e comunicação com Fortran e C, vetorização, álgebra linear e geração de números aleatórios. O NumPy constitui uma **fundação sólida** para muitas ferramentas aplicadas em análise de dados — inclusive o Pandas, que virá nas aulas seguintes.

A razão de sua popularidade é enunciada de forma enfática: sua **eficiência quando comparado às listas**. É muito mais rápido trabalhar com NumPy, porque o pacote permite o acesso a dados de modo muito mais eficiente.

7.2 Estruturas de dados: vetores e matrizes

Um **vetor** numérico em Python é uma sequência indexada de 0 em diante, com a restrição fundamental de que todos os elementos são **de um mesmo tipo**. Essa homogeneidade é justamente o que permite o armazenamento contíguo em memória e as otimizações de baixo nível.

Uma **matriz** organiza os dados em duas dimensões, com índices de linha e coluna. A hierarquia completa apresentada é: **escalares, vetores, matrizes, tensores** — respectivamente estruturas de zero, uma, duas e três ou mais dimensões.

7.3 Criação de arrays

Os slides apresentam os construtores principais:

```
import numpy as np

t1 = np.ones((4, 3, 2))            # tensor 3D preenchido com 1
t2 = np.zeros((4, 3, 2))           # tensor 3D preenchido com 0
t3 = np.random.random((4, 3, 2))   # valores aleatorios entre 0 e 1

X = np.array([[1, 2], [3, 4], [5, 6]])
v = np.array([1, 2, 3, 4, 5, 6])
```

O atributo `shape` retorna a forma da estrutura: para os tensores acima, `t1.shape` devolve `(4, 3, 2)`; para a matriz `X`, `X.shape` devolve `(3, 2)`. Compreender `shape` é pré-requisito para tudo o que vem depois, porque quase todo erro em álgebra linear computacional é um erro de dimensão.

Além de `shape`, a aula cobre os demais **atributos de um NumPy array** e um conjunto de **funções para arrays**.

7.4 Operações

Aqui está o ponto conceitual mais importante da aula: a distinção entre operações **elemento a elemento** e operações de **álgebra linear**. Considerando as matrizes

```
A = [[1, 2],      B = [[10, 20],
     [4, 5]]           [30, 40]]
```

os três tipos de operação produzem resultados distintos:

- **Multiplicação elemento a elemento**, `A * B`, resulta em `[[10, 40], [120, 200]]` — cada posição multiplicada por sua correspondente.
- **Multiplicação matricial**, `A.dot(B)`, resulta em `[[70, 100], [190, 280]]` — o produto de linhas por colunas da álgebra linear.

- **Multiplicação por escalar**, `2 * A`, resulta em `[[2, 4], [8, 10]]`.

Confundir `*` com `.dot()` é uma das fontes mais comuns de erros silenciosos em código numérico: ambos executam sem erro, mas produzem resultados completamente diferentes.

7.5 Exercício de desempenho

O primeiro exercício testa empiricamente a afirmação de que operações vetorizadas são extremamente mais eficientes que loops. A tarefa é criar dois vetores com 100000 elementos e fazer um produto escalar (*dot product*), comparando lista nativa contra array NumPy. A dica é usar `time.process_time()` do pacote `time` para salvar o tempo antes e depois da operação, sendo o tempo decorrido em segundos a diferença entre esses valores.

```
import time
import numpy as np

a = 100_000 * [1]
b = 100_000 * [1]

inicio = time.process_time()
resultado = sum(x * y for x, y in zip(a, b))
print(time.process_time() - inicio)

a = np.asarray(a)
b = np.asarray(b)

inicio = time.process_time()
resultado = a.dot(b)
print(time.process_time() - inicio)
```

O aprendizado não é apenas que NumPy é mais rápido, mas **por que**: o loop Python interpreta cada iteração individualmente, enquanto a operação vetorizada delega o laço a código compilado.

7.6 Estudo de caso: análise de demonstração financeira

O segundo exercício coloca o aluno no papel de um cientista de dados em uma empresa de consultoria, cujo colega do departamento de auditoria pediu ajuda para avaliar o demonstrativo financeiro de uma organização X. São fornecidos dois vetores com dados mensais de um ano:

```
import numpy as np

receitas = np.array([14574.49, 7606.46, 18611.41, 19175.41, 8758.65,
                     8105.44, 11496.28, 9766.09, 10305.32, 18379.96,
                     10713.97, 15433.50])

custos = np.array([12051.82, 5695.07, 12319.20, 12089.72, 8658.57,
                   840.20, 3285.73, 5821.12, 6976.93, 16618.61,
                   10054.37, 3803.96])
```

As métricas financeiras a calcular são:

- O **lucro** de cada mês;

- O **lucro após imposto** para cada mês, sendo o imposto de 30% sobre o lucro;
- A **margem de lucro** de cada mês, isto é, o lucro depois de descontado o imposto dividido pela receita;
- Os **meses bons**, onde o lucro após taxação foi maior que o lucro médio do ano, usando a função `np.where`;
- Os **meses ruins**, o contrário dos meses bons;
- O **melhor mês** (descontado imposto);
- O **pior mês** (descontado imposto).

As funções sugeridas são `mean()`, `max()`, `min()` e `np.where()`.

```
lucro = receitas - custos
lucro_liquido = lucro * 0.7
margem = lucro_liquido / receitas

meses_bons = np.where(lucro_liquido > lucro_liquido.mean())
melhor_mes = np.where(lucro_liquido == lucro_liquido.max())
```

O exercício é uma síntese perfeita da aula: nenhuma das operações acima usa um único `for`. Subtração de vetores, multiplicação por escalar, divisão elemento a elemento e filtragem condicional são todas expressas em uma linha cada, exatamente como a vetorização promete.

8 Funções e Dataframes

Material de slides não disponível para esta aula.

Esta aula articula dois temas: a retomada de **funções** e a introdução aos **DataFrames**. O material de apoio é composto pelos mesmos notebooks de funções e bibliotecas (`Funcoes_e_Bibliotecas.ipynb` e as versões feitas em aula), o que indica uma abordagem de laboratório em que o DataFrame é apresentado ao vivo.

Do que o restante do material permite estabelecer: o DataFrame é a estrutura tabular fornecida pela biblioteca **Pandas**, sucessora natural do array NumPy quando os dados deixam de ser homogêneos e passam a ter colunas nomeadas e de tipos distintos. A conexão com a aula anterior é direta — o array NumPy é homogêneo e indexado por posição; o DataFrame permite tipos diferentes por coluna e indexação por rótulo, mantendo por baixo a mesma eficiência vetorizada.

O único código Pandas explicitamente presente no material de slides da disciplina aparece na aula de LLM para Programação, e serve como ilustração mínima do fluxo típico com DataFrames:

```
import pandas as pd

df = pd.read_csv('vendas.csv')
resultado = df.groupby('produto')['valor'].sum()
resultado = resultado.sort_values(ascending=False)
resultado.to_csv('total_vendas.csv')
```

As quatro operações desse trecho — **leitura**, **agregação por grupo**, **ordenação** e **exportação** — constituem o núcleo do trabalho cotidiano com DataFrames, e reaparecem nas aulas de manipulação de dados e nos estudos de caso.

9 Manipulação de dados e noções de estatística com Python

Material de slides não disponível para esta aula.

Esta aula é dedicada à manipulação de dados e a noções de estatística. Os arquivos de apoio revelam o escopo: notebooks de operações com DataFrames e criação de gráficos, notebooks específicos de manipulação de dados e noções de estatística, um notebook de estudo de caso do Titanic, além de bases de dados em CSV — `FuelEfficiency.csv`, `results_modif.csv` e `Titanic_dataset.csv` — e um desafio de primeiros passos.

O material de slides disponível na disciplina sustenta alguns pilares deste conteúdo.

9.1 Estatística descritiva e simulação

A base estatística já foi lançada no bloco introdutório de Python, com o exercício da **Lei dos Grandes Números** e a distribuição normal de média 0 e desvio padrão 1, onde aproximadamente 68,2% das observações caem no intervalo de -1 a 1. As funções de resumo `mean()`, `max()` e `min()`, exercitadas sobre arrays NumPy no estudo financeiro, são as mesmas usadas sobre colunas de um DataFrame.

9.2 Visualizações

Os tipos de gráfico cobertos na disciplina, listados no material da última aula sob o título "É bom lembrar!", são: **histograma**, **densidade**, **boxplot**, **gráfico em barras**, **gráfico de dispersão** e **matriz de correlação**. Cada um responde a uma pergunta distinta: histograma e densidade descrevem a distribuição de uma variável; o boxplot resume distribuição e destaca outliers, sendo especialmente útil para comparar grupos; o gráfico de barras compara categorias; o de dispersão examina a relação entre duas variáveis contínuas.

9.3 Correlação

Sobre correlação, o material é explícito e cauteloso. A correlação indica a **força e a direção** do relacionamento entre dois atributos. Trata-se de uma medida da relação entre dois atributos, embora **correlação não implique causalidade**: duas variáveis podem estar altamente correlacionadas sem que exista relação de causa e efeito entre elas. Na análise de correlação linear, o objetivo é determinar o grau de relacionamento entre duas variáveis.

Essa ressalva é a ponte entre a estatística descritiva e a inferência responsável, e reaparece com força nos cuidados éticos discutidos na aula final.

10 Análise de dados com Python via estudos de casos

Material de slides não disponível para esta aula.

Esta aula aplica o ferramental acumulado em estudos de caso. Os arquivos de apoio são os mesmos da aula anterior, com destaque para os notebooks `EstudodeCaso_titanic.ipynb` e sua versão feita em aula, acompanhados de `Titanic_dataset.csv`, além da base `FuelEfficiency.csv` e do desafio de primeiros passos.

O formato de estudo de caso é o mesmo já praticado ao longo da disciplina: parte-se de uma base real, define-se um conjunto de perguntas de negócio ou de pesquisa, e o código é escrito para respondê-las. O exercício de **análise de demonstração financeira** da aula de NumPy é o protótipo desse formato em escala reduzida — cenário de negócio, dados fornecidos e uma lista explícita de métricas a calcular.

Os estudos de caso documentados nos slides da disciplina, e que caracterizam este tipo de trabalho, são apresentados na aula final: a base de **medicamentos para ansiedade** (`Islander_data.csv`) e a base de **tendências mundiais** (`TendenciasMundiais.csv`). Ambos são descritos em detalhe na seção seguinte.

O roteiro geral de um estudo de caso, tal como o material permite reconstruir, envolve: carregar a base em um DataFrame, produzir um resumo estatístico das colunas, verificar valores nulos, inconsistências e possíveis outliers, gerar visualizações apropriadas ao tipo de cada variável, investigar associações entre variáveis e, por fim, documentar os achados em linguagem clara.

11 LLM Para Programação

Esta é a aula de fechamento, e seus objetivos são três: **compreender copilotos**, entendendo o que são copilotos de código e como LLMs podem auxiliar no desenvolvimento Python; **conhecer ferramentas**, explorando as principais opções disponíveis no mercado para programação assistida por IA; e **aplicar na prática**, dominando técnicas de prompt e praticando casos reais de uso em projetos Python.

11.1 O que é um copiloto de código

Um **copiloto de código** é um assistente de IA que ajuda programadores a escrever, entender e melhorar código. Utilizando modelos de linguagem avançados, esses sistemas podem:

- Gerar código a partir de descrições em linguagem natural
- Sugerir completações e melhorias em tempo real
- Explicar trechos complexos de código
- Identificar e corrigir bugs
- Refatorar código para melhor legibilidade

A tese central é enunciada sem ambiguidade: **copiloto não é substituição** — é uma parceria que potencializa as habilidades do programador.

11.2 O legado: como se buscava ajuda antes dos LLMs

Antes dos copilotos, a assistência no desenvolvimento vinha de fontes tradicionais que exigiam pesquisa manual e adaptação de soluções:

- **Stack Overflow**, a maior comunidade global de perguntas e respostas. Buscava-se um erro, encontrava-se uma solução semelhante e adaptava-se. Exigia boa formulação de perguntas e pesquisa eficaz.
- **Documentação oficial** — docs do Python, do Pandas, de APIs. Era 100% manual: leitura, interpretação, tentativa e erro. Essencial para aprofundamento, mas demorada para resolução rápida.
- **Tutoriais e fóruns**, como GeeksForGeeks, Medium, W3Schools e Real Python. Fontes ricas de conhecimento e exemplos, mas exigiam curadoria e comparação entre abordagens.
- **Exemplos no GitHub**, procurando repositórios com problemas semelhantes, copiando trechos e adaptando-os, com necessidade de testar e integrar cuidadosamente.
- **Ajuda de colegas**, via pair programming ou consultoria com desenvolvedores experientes. Extremamente eficaz para aprendizado e problemas complexos, mas dependente da disponibilidade de outras pessoas.

A nova era, segundo os slides, unifica tudo isso: o copiloto busca, explica, sugere e até depura, sem exigir alternância constante entre Stack Overflow, documentação e blogs.

11.3 Principais ferramentas

As ferramentas citadas, com suas caracterizações:

- **ChatGPT** — conversa interativa para gerar, explicar e depurar código Python com contexto completo.
- **GitHub Copilot** — autocomplete inteligente integrado ao VS Code e a IDEs populares.
- **Cursor** — editor de código com IA nativa, comandos em linguagem natural e edição contextual.
- **Jupyter AI** — extensões de IA para notebooks Jupyter, ideal para ciência de dados e análise.
- **Gemini no Colab** — IA integrada ao Google Colab, capaz de gerar células, explicar código e auxiliar em experimentos de ciência de dados.
- **Windsurf** — IDE AI-first semelhante ao Cursor, com edição contextual e comandos de linguagem natural aplicados ao projeto inteiro.
- **CodeWhisperer** — copiloto da AWS com sugestões de código, integração para cloud e suporte forte para Python e Java no VS Code.
- **Google AI Studio** — ambiente para criar, testar e integrar modelos Gemini, com geração de código para aplicações em Python e JavaScript.

11.4 Quatro tipos de ferramenta

A taxonomia proposta é útil porque associa cada categoria a um momento de uso.

Assistente geral de código. Ferramenta que gera, explica e depura código fora da IDE, sem entender a estrutura completa do projeto. Exemplos: ChatGPT, Gemini, Claude. Usa-se quando se precisa de ajuda conceitual, rascunho de código, explicação de erros ou revisão de lógica.

Copiloto de código. Atua dentro da IDE, sugerindo código em tempo real, completando funções, explicando trechos e ajudando no fluxo de escrita. Exemplos: GitHub Copilot, CodeWhisperer, ChatGPT for VS Code. Usa-se quando já se está programando e se quer acelerar a escrita, refatorar, criar testes e receber sugestões.

Vibe coding ou IDE AI-first. Ambiente onde a IA entende todo o projeto, cria múltiplos arquivos, modifica arquitetura e executa tarefas grandes apenas com linguagem natural. Exemplos: Cursor IDE, Windsurf. Usa-se quando se quer criar ou modificar projetos completos rapidamente, indo além de apenas completar código.

IA integrada a notebooks. IA embutida em Jupyter ou Colab para gerar células, explicar código, fazer análises, criar visualizações e ajudar em ciência de dados. Exemplos: Jupyter AI, Gemini no Google Colab. Usa-se ao trabalhar com dados, experimentar modelos, prototipar análises ou criar notebooks educacionais.

11.5 Padrões de prompt eficazes

Três princípios organizam a construção de um bom prompt para geração de código:

- **Seja específico** — descreva claramente o problema, incluindo entrada esperada, saída desejada e restrições.
- **Forneça contexto** — mencione bibliotecas, versão do Python e estruturas de dados em uso.
- **Defina formato** — peça o código no formato desejado: função, classe, script completo ou snippet.

O exemplo de prompt eficaz apresentado é: *"Crie uma função Python que receba uma lista de números e retorne apenas os pares, usando list comprehension."* Note que o prompt especifica entrada (lista de números), saída (apenas os pares) e restrição de implementação (list comprehension).

11.6 Quatro casos de uso demonstrados

Gerar script. Enunciado: "Preciso de um script que leia um arquivo CSV com dados de vendas, calcule o total de vendas por produto e exporte um novo CSV com os resultados ordenados." O código gerado:

```
import pandas as pd

df = pd.read_csv('vendas.csv')
resultado = df.groupby('produto')['valor'].sum()
resultado = resultado.sort_values(ascending=False)
resultado.to_csv('total_vendas.csv')
```

O copiloto entendeu a tarefa e gerou código limpo com Pandas, incluindo leitura, agregação, ordenação e exportação.

Explicar código. Dado o trecho `result = [x**2 for x in range(10) if x % 2 == 0]`, a explicação para iniciantes é que o código usa list comprehension para criar uma lista, percorrendo números de 0 a 9, filtrando apenas os pares e elevando cada um ao quadrado, com resultado `[0, 4, 16, 36, 64]`. Copilotos transformam código complexo em linguagem acessível, acelerando o aprendizado.

Debug guiado. O problema apresentado tem três etapas:

```
# 1. Problema
def calcular_media(numeros):
    return sum(numeros) / len(numeros)

calcular_media([])   # ZeroDivisionError

# 3. Solucao corrigida
def calcular_media(numeros):
    if not numeros:
        return 0
    return sum(numeros) / len(numeros)
```

O diagnóstico da IA: a função falha ao receber uma lista vazia porque tenta dividir por zero; é necessário adicionar validação de entrada.

Refatoração. O objetivo é melhorar legibilidade e performance.

```
# Antes: codigo confuso, nomes genericos, dificil manutencao
def p(l):
    r = []
    for i in l:
        if i > 0:
            r.append(i * 2)
    return r

# Depois: claro, documentado, pythonico
def duplicar_positivos(numeros):
    """Retorna lista com numeros positivos duplicados"""
    return [num * 2 for num in numeros if num > 0]
```

11.7 Boas práticas e cuidados

Quatro recomendações fecham a parte conceitual:

- **Sempre revise.** Código gerado por IA pode conter erros sutis ou não seguir as melhores práticas da sua equipe.
- **Teste tudo.** Valide com casos de teste, incluindo edge cases que a IA pode não ter considerado.
- **Privacidade.** Nunca envie dados sensíveis, credenciais ou informações proprietárias para copilotos públicos.
- **Continue aprendendo.** Use copilotos para acelerar, não para substituir seu desenvolvimento como programador.

A síntese da aula: **copiloto é parceiro, não piloto automático**.

11.8 Estudo de caso: medicamentos para ansiedade

O estudo de caso principal usa dados de um experimento sobre os efeitos de medicamentos para ansiedade e seu impacto em grupos que têm majoritariamente lembranças felizes ou tristes. A fonte é a base pública disponível no Kaggle (`memory-test-on-drugged-islanders-data`).

A codificação das substâncias e dosagens é:

- **A** — Alprazolam (Xanax, longo prazo): 1mg, 3mg, 5mg
- **T** — Triazolam (Halcion, curto prazo): 0,25mg, 0,5mg, 0,75mg
- **S** — Sugar Tablet (placebo): 1 tablete, 2 tabletes, 3 tabletes

As colunas do dataset são:

Coluna	Descrição
`first_name`, `last_name`	Nome e sobrenome fictícios do participante, apenas identificadores, não usados na análise estatística
`age`	Idade do participante, em anos
`Happy_Sad_group`	Grupo emocional de origem: H para pessoas com maior parte de lembranças felizes, S para lembranças tristes
`Drug`	Substância administrada: A, T ou S
`Dosage`	Nível de dose (1, 2 ou 3)
`Mem_Score_Before`	Pontuação no teste de memória antes do medicamento
`Mem_Score_After`	Pontuação no teste de memória depois do medicamento
`Diff`	Diferença entre depois e antes; positivo indica melhora, negativo indica piora

A tarefa proposta é carregar a base `Islander_data.csv` e fazer, livremente, análises gráficas para entendimento dos dados utilizando qualquer copiloto ou assistente.

11.9 Como um copiloto ajuda na análise exploratória

Cinco contribuições são enumeradas:

- **Carga e limpeza de dados** — sugere código pronto para importar, transformar e preparar os dados, acelerando a etapa inicial.
- **Visualização automática** — propõe e gera gráficos relevantes (histograma, boxplot, dispersão) para explorar a distribuição e as relações das variáveis.
- **Identificação de padrões** — explica padrões, outliers e correlações complexas, fornecendo insights rapidamente.
- **Documentação simples** — ajuda a documentar achados e processo em linguagem clara, ideal para relatórios.
- **Pipeline de EDA guiado** — auxilia mesmo quem não tem experiência em programação a construir um pipeline completo de Análise Exploratória de Dados.

A ressalva é permanente: a IA atua como analista assistente, fornecendo sugestões e explicações, mas **a validação e o pensamento crítico são sempre seus**.

11.10 Padrões de prompt para EDA

Cinco prompts modelo são fornecidos para conduzir uma análise exploratória:

1. "Carregue o arquivo e mostre um resumo estatístico das colunas."
2. "Crie gráficos que ajudem a entender a distribuição dos resultados."
3. "Mostre valores nulos, inconsistências e possíveis outliers."
4. "Explique em linguagem simples o que o gráfico significa."
5. "Quais variáveis parecem mais associadas ao desempenho nos testes de memória?"

Com prompts claros e objetivos, o copiloto atua como assistente de dados, permitindo que qualquer pessoa, mesmo sem profundo conhecimento de programação, acompanhe e realize análises.

11.11 Cuidados ao analisar dados humanos

A análise de dados humanos, especialmente na área da saúde, exige rigor ético e metodológico. Cinco cuidados são explicitados:

- **Conclusões clínicas.** Não tire conclusões clínicas apenas com Análise Exploratória de Dados; ela é um primeiro passo, não um diagnóstico.
- **Interpretações causais.** Evite afirmar causalidade, como "a droga X melhorou a ansiedade", sem estudos clínicos apropriados e controlados.
- **Atenção a vieses.** Mantenha vigilância sobre vieses potenciais — idade, gênero, dose, baseline emocional — que podem distorcer resultados.
- **Privacidade de dados.** Nunca envie dados pessoais identificáveis ou informações sensíveis de pacientes para copilotos de IA públicos.
- **Julgamento clínico.** Trate todos os dados como agnósticos, sem qualquer juízo clínico pessoal ou preconceito.

11.12 Revisão de gráficos e correlação

A aula retoma os tipos de gráfico já estudados: **histograma**, **densidade**, **boxplot**, **gráfico em barras** e **gráfico de dispersão**. E reafirma o conceito de **matriz de correlação**: a correlação indica a força e a direção do relacionamento entre dois atributos, embora correlação não implique causalidade — duas variáveis podem estar altamente correlacionadas sem que haja relação de causa e efeito. Na análise de correlação linear, o objetivo é determinar o grau de relacionamento entre duas variáveis.

11.13 Estudo de caso extra: tendências mundiais

Um segundo estudo de caso propõe carregar a base `tendenciasMundiais.csv` em um DataFrame e mostrar graficamente a relação entre **Expectativa de Vida** e **Taxa de Fertilidade** por país para os anos de 1960 e 2013, ambos em um mesmo gráfico, utilizando a função `scatter`. Pede-se comparar e avaliar os dois anos, incluindo legenda, título e nomes dos eixos, e há convite explícito para criar outros gráficos usando os tipos aprendidos — boxplot, violinplot, histograma e outros.

O exercício é elegante porque um único gráfico de dispersão com dois anos sobrepostos torna visível, de imediato, um dos fenômenos demográficos mais importantes do século: o deslocamento conjunto de países para maior expectativa de vida e menor fertilidade.

12 Síntese da Disciplina

A DPIA constrói um percurso que vai do conceito ao código e do código de volta ao conceito, agora mediado por IA. Vale reconstituir esse arco.

A **aula 1** estabelece que Inteligência Artificial, Machine Learning e Deep Learning são círculos concêntricos, e que o salto do ML clássico para o DL foi motivado por um problema concreto: a impossibilidade de especificar manualmente as características relevantes em domínios como visão computacional. A história dos invernos e booms ensina a calibrar expectativas, e os seis equívocos comuns ensinam a organizar equipes e projetos — a preferência pelo simples sobre o complexo, a necessidade de conhecimento de domínio e o caráter exploratório dos projetos são lições que se aplicam a qualquer trabalho de dados.

As **aulas 2 a 5** constroem a base de programação. Elas começam pela lógica e pelo algoritmo, passam pelos tipos de dados — com a distinção fundamental entre estruturas mutáveis e imutáveis, ordenadas e não ordenadas, indexadas por posição e indexadas por chave — e chegam às estruturas de controle e às funções. Duas escolhas pedagógicas se destacam: pedir a solução dos mesmos exercícios **com e sem estruturas de repetição**, o que antecipa a vetorização, e usar a **Lei dos Grandes Números** como fio condutor, aparecendo primeiro como script, depois como função rígida e finalmente como função parametrizada com argumentos opcionais.

A **aula 6** faz a transição de listas para arrays. O NumPy é apresentado não como mais uma biblioteca, mas como a fundação sobre a qual todo o ecossistema de análise de dados em Python está construído. Os conceitos de `shape`, de hierarquia escalar-vetor-matriz-tensor e a distinção entre multiplicação elemento a elemento e multiplicação matricial são pré-requisitos que reaparecerão em toda modelagem posterior. O estudo da demonstração financeira demonstra a vetorização em uso real: sete métricas de negócio calculadas sem um único laço explícito.

As **aulas 7 a 9** levam esse ferramental para dados tabulares heterogêneos, via DataFrames, estatística descritiva e visualização. Aqui se estabelece o vocabulário gráfico da disciplina — histograma, densidade, boxplot, barras, dispersão e matriz de correlação — e uma advertência que atravessa todo o curso: correlação não implica causalidade.

A **aula 10** fecha o círculo. Tendo aprendido a escrever o código à mão, o aluno aprende a colaborar com LLMs. A taxonomia de ferramentas — assistente geral, copiloto na IDE, IDE AI-first e IA em notebooks — permite escolher a ferramenta certa para cada momento. Os padrões de prompt (ser específico, fornecer contexto, definir formato) tornam a colaboração previsível. E os quatro casos de uso demonstrados — gerar, explicar, depurar e refatorar — mapeiam exatamente as tarefas que ocuparam as aulas anteriores.

A conexão mais importante entre as pontas do curso é ética e epistêmica. A aula 1 alertava que os melhores modelos não são necessariamente os mais avançados e que projetos de dados são exploratórios; a aula 10 alerta que o copiloto é parceiro e não piloto automático, que todo código gerado deve ser revisado e testado, e que dados humanos exigem cuidado redobrado com causalidade,

vieses e privacidade. É a mesma disposição intelectual em dois contextos: **a ferramenta acelera, mas o julgamento crítico permanece humano**.

O aluno que conclui a disciplina sai capaz de ler e escrever Python idiomático, manipular arrays e DataFrames com operações vetorizadas, produzir uma análise exploratória completa com visualizações apropriadas, interpretar correlações sem confundi-las com causalidade e usar copilotos de IA de forma produtiva e responsável. Mais do que qualquer sintaxe específica, o que a DPIA entrega é a desmistificação anunciada no título: a percepção de que programar IA é uma habilidade acessível, construída sobre poucos princípios bem compreendidos e muita prática deliberada.