EAE1106 - Métodos Computacionais para Economia

Faculdade de Economia, Administração, Contabilidade e AtuariaUniversidade de São Paulo

Arthur Viaro

20 de fevereiro, 2026

Índice

Sobre este material 3
Como usar . . . 3

Prefácio 4

I Fundamentos de Computação 6

1 Como um computador funciona 9
1.1 Introdução . . . 9
1.1.1 Pré-requisitos . . . 9
1.2 Como um computador funciona . . . 9
1.2.1 Sistema computacional . . . 11
1.2.2 Arquitetura de um computador . . . 12
1.2.3 Bits e bytes . . . 13
1.2.4 ASCII . . . 14
1.2.5 UNICODE . . . 15
1.2.6 RGB . . . 16
1.2.7 Processamento, memória e armazenamento na prática . . . 17
1.3 Arquivos e diretórios . . . 18
1.3.1 Estrutura hierárquica . . . 19
1.3.2 Extensões de arquivos . . . 19
1.4 Terminal . . . 22
1.5 Linguagens de programação . . . 23
1.5.1 Sintaxe e semântica . . . 24
1.5.2 Diferenças entre linguagens naturais e formais . . . 24
1.5.3 Alto nível X baixo nível . . . 25
1.5.4 Linguagem Compilada X Interpretada . . . 25
1.5.5 Paradigmas . . . 26
1.5.6 Tipagem . . . 26
1.6 Algoritmos . . . 27
1.7 Pseudocódigo . . . 27
1.8 Conclusão . . . 28
1.9 Exercícios . . . 28

2 Primeiros passos no Python **31**
2.1 Introdução 31
2.1.1 Pré-requisitos 32
2.2 O Que é Python? 32
2.2.1 Por que Python em um Curso de Economia? 33
2.3 Como Instalar o Python? 34
2.3.1 Instalação pelo Site Oficial 34
2.3.2 Instalando via Anaconda (Recomendado) 37
2.3.3 Conhecendo o Anaconda 38
2.3.4 Gerenciamento de ambientes 40
2.3.5 Instalando Pacotes 43
2.4 IDEs 44
2.4.1 Componentes típicos de uma IDE 44
2.4.2 Spyder 45
2.4.3 Jupyter Notebook 45
2.4.4 Google Colaboratory (Colab) 47
2.4.5 VS Code 48
2.4.6 Positron 49
2.4.7 Qual Escolher? 50
2.5 Markdown 51
2.6 Construindo o Primeiro Programa 52
2.6.1 Usando Python como uma Calculadora 52
2.6.2 Instruções e variáveis 54
2.6.3 Valores e Tipos 56
2.7 Conclusão 57
2.8 Exercícios 57

3 Tipos de Dados e Expressões **59**
3.1 Introdução 59
3.1.1 Pré-requisitos 59
3.2 Valores, Tipos e Objetos 59
3.3 Breve Introdução às Funções 60
3.3.1 Argumentos 60
3.3.2 Variáveis 61
3.4 Tipos Básicos 61
3.4.1 Inteiros (int) 61
3.4.2 Pontos Flutuantes (float) 62
3.4.3 Strings (str) 63
3.4.4 Booleanos (bool) 69
3.4.5 Listas 70
3.4.6 Tuplas 76
3.4.7 Conjuntos 81
3.4.8 Dicionários 86

3.5 Explorando o Python . . . 93
3.6 Conclusão . . . 96
3.7 Exercícios . . . 96

4 Controle de fluxo e iteração 97
4.1 Controle de Fluxo . . . 97
4.1.1 Expressões Booleanas . . . 97
4.1.2 Instruções `if` . . . 99
4.1.3 Controle de Fluxo: `elif` e `else` . . . 99
4.1.4 Operador `or` . . . 101
4.1.5 Operador `and` . . . 102
4.1.6 Instrução `match` . . . 103
4.2 Estruturas de Repetição . . . 105
4.2.1 Repetições com `while()` . . . 105
4.2.2 Repetições com `for()` . . . 107
4.2.3 Mais sobre Listas . . . 109
4.2.4 Interrompendo um Laço de Repetição . . . 111
4.2.5 Aplicação: Teorema Central do Limite . . . 112
4.3 Exercícios . . . 114

5 Funções 116
5.1 Definindo uma função . . . 116
5.2 Parâmetros . . . 117
5.2.1 Valores padrão . . . 117
5.2.2 Parâmetros posicionais e nomeados . . . 118
5.2.3 Exemplo: juros compostos . . . 118
5.3 Retornando valores . . . 119
5.3.1 Exemplo: encadeando cálculos . . . 120
5.3.2 Múltiplos retornos . . . 122
5.3.3 Exemplo: métricas de inflação . . . 123
5.4 Aplicação: Teorema Central do Limite . . . 124
5.4.1 Verificando a convergência . . . 125
5.5 Docstrings . . . 125
5.6 Escopo de variáveis . . . 126
5.6.1 Escopo local . . . 127
5.6.2 Escopo enclosing . . . 127
5.6.3 Escopo global . . . 128
5.6.4 Escopo enclosing com `nonlocal` . . . 129
5.6.5 Escopo built-in . . . 130
5.7 args e kwargs . . . 130
5.7.1 `*args`: múltiplos argumentos posicionais . . . 131
5.7.2 `**kwargs`: múltiplos argumentos nomeados . . . 132
5.7.3 Combinando `*args` e `**kwargs` . . . 134

5.8 Funções anônimas (lambda) . . . 134
5.8.1 Função map() . . . 135
5.8.2 Função filter() . . . 136
5.8.3 Função sorted() . . . 137
5.9 Exercícios . . . 139

6 Exceções 141
6.1 Os erros mais comuns em Python . . . 141
6.2 Erros de sintaxe (SyntaxError) . . . 142
6.3 Erros em tempo de execução . . . 143
6.3.1 ValueError . . . 143
6.4 try e except . . . 143
6.4.1 O problema do NameError com try/except . . . 144
6.5 else . . . 144
6.5.1 Loop de validação com else . . . 145
6.5.2 Utilizando funções . . . 146
6.6 pass . . . 147
6.6.1 Exemplo Prático: Limpeza de base de dados . . . 149
6.7 raise . . . 150
6.8 Capturando múltiplas exceções . . . 150
6.9 Exercícios . . . 151

7 Expressões Regulares 153
7.1 O módulo re . . . 153
7.2 Sintaxe dos padrões . . . 154
7.2.1 Metacaracteres de correspondência . . . 154
7.2.2 Metacaracteres de quantidade . . . 155
7.2.3 Âncoras, grupos e controle . . . 155
7.3 re.search . . . 156
7.4 Agrupamento em Expressões Regulares . . . 156
7.5 re.match . . . 158
7.6 re.findall . . . 159
7.7 re.finditer . . . 161
7.8 re.split . . . 162
7.9 re.sub . . . 163
7.10 Flags . . . 164
7.11 Exercícios . . . 165

References 167

Sobre este material

Este website reúne o material didático da disciplina **EAE1106 — Métodos Computacionais para Economia**, ministrada no primeiro semestre de 2026 para as turmas do período noturno no Departamento de Economia da Universidade de São Paulo (USP).

O conteúdo foi baseado nas notas de aulas e materiais desenvolvidos pelos professores Danilo Souza e Claudio Lucinda e está em constante atualização. Expresso meu especial agradecimento ao Prof. Danilo por compartilhar suas notas de aula e pelos valiosos feedbacks durante a preparação deste material.

Dica: Recomendo fortemente a consulta ao livro digital preparado pelo Prof. Danilo Souza e utilizado nas turmas do período diurno neste semestre.

Sugestões de melhorias são sempre bem-vindas, seja por meio de issues no GitHub ou pelo e-mail: arthur.viaro@gmail.com

Como usar

Este material foi concebido como um livro digital interativo para apoiar as aulas presenciais e servir como referência permanente. Mais do que um texto estático, ele é um ambiente de experimentação. Ler código raramente é suficiente para desenvolver fluência. A compreensão efetiva dos conceitos depende da experimentação ativa. Por isso, encorajamos você a rodar os exemplos em seu próprio ambiente, modificando comandos e testando variações.

Embora os capítulos sigam a sequência lógica da disciplina, o material permite uma consulta flexível e não linear. Cada seção é composta por três elementos fundamentais:

- Exemplos práticos dos conceitos teóricos.
- Observações sobre boas práticas, detalhes técnicos e "armadilhas" comuns.
- Exercícios para consolidar o aprendizado.

Este material não substitui a presença em sala de aula. Ao final do semestre, espera-se que você tenha autonomia para utilizar a programação como uma ferramenta poderosa para organizar dados, explorar informações e fundamentar análises empíricas em Economia.

Arthur Viaro Prof. Assistente em Economia FEA-USP, Fevereiro 2026

Prefácio

Este material foi desenvolvido como apoio didático para a disciplina EAE1106 – Métodos Computacionais para Economia, ministrada pelo Departamento de Economia da Universidade de São Paulo. A disciplina tem por objetivo oferecer aos estudantes uma introdução sistemática à lógica de programação e às técnicas de análise de dados utilizando linguagens de programação, com ênfase em **Python** e uma breve introdução a **R** ao final do semestre.

O curso parte da motivação de que grande parte das perguntas contemporâneas em economia – desde a exploração de grandes bases de dados até a implementação de simulações e modelos numéricos – exige o uso de ferramentas computacionais para organizar, manipular, visualizar e interpretar dados de forma eficiente. Consequentemente, a capacidade de expressar uma sequência de operações de forma reproduzível e automatizada é uma competência essencial para economistas que desejam conduzir análises empíricas rigorosas e comunicar resultados de forma clara e estruturada.

Este material foi concebido com duas orientações principais:

- **Progressão pedagógica gradual**. O conteúdo inicia por fundamentos de lógica de programação, tipos de dados e objetos dentro da linguagem, controle de fluxo e funções, e evolui para tópicos aplicados de manipulação de dados, álgebra linear, visualização e análise integrada de dados. A estrutura dos capítulos foi planejada para acompanhar as etapas do raciocínio computacional e da análise empírica.
- **Aplicação orientada a problemas econômicos**. Sempre que possível, os exemplos e exercícios são contextualizados em questões empíricas de interesse econômico, tais como manipulação e análise de bases de dados administrativas, cálculo de estatísticas descritivas e visualização exploratória. Isso visa reforçar a conexão entre métodos computacionais e aplicações reais em pesquisa econômica.

Além dessas orientações gerais, o curso está organizado em três grandes blocos de conteúdo, que estruturam tanto as aulas quanto este material:

1. **Alfabetização computacional e lógica básica**, que introduz os conceitos fundamentais de computação, a interação entre linguagens de programação e modelos de inteligência artificial, tipos primitivos e objetos básicos dentro do Python, além de como implementar controle de fluxo, iteração, funções personalizadas dentro da linguagem.

2. **Computação numérica, análise de dados e visualização**, com foco em estruturas matriciais e álgebra linear através do uso do pacote `NumPy`, manipulação eficiente de dados em dataframes com `Pandas`, construção e personalização de gráficos, e princípios básicos de visualização de dados, culminando em uma análise empírica integrada que enfatiza organização de projetos, reprodutibilidade e construção de pipelines completos.
3. **Temas complementares**, que incluem uma pequena introdução a tópicos mais avançados, como a programação orientada a objetos, e uma breve introdução à linguagem R, com o objetivo de familiarizar o estudante com outro ecossistema amplamente utilizado em análise empírica e principal linguagem utilizada ao longo da sequência de cursos obrigatórios de Econometria.

De forma geral, o conteúdo foi organizado como um livro digital interativo, com exemplos de código e saídas esperadas, visualizações e anotações integradas. O uso de linguagens de programação neste formato permite que o leitor aprenda de forma construtiva e ativa, facilitando a compreensão da sintaxe, lógica e aprendendo a avaliar erros e como corrigí-los. Espero que este material seja útil tanto para estudantes que se aproximam pela primeira vez da programação aplicada à economia quanto para aqueles que desejam reforçar e aplicar os conceitos em trabalhos empíricos e projetos de pesquisa.

Parte I

Fundamentos de Computação

Esta parte inicial do curso é dedicada à alfabetização computacional e à construção dos fundamentos da lógica de programação. O objetivo é desenvolver, de forma gradual, a capacidade de compreender, escrever e interpretar código, preparando o terreno para aplicações empíricas mais avançadas nas etapas seguintes do curso.

Nesta etapa, trabalharemos com os conceitos básicos da linguagem Python, sem assumir conhecimento prévio em programação. O foco está menos em bibliotecas especializadas e mais na compreensão dos elementos fundamentais que estruturam qualquer programa: sintaxe, tipos de dados e objetos, operações, controle de fluxo, funções e boas práticas de código. Listo abaixo alguns dos principais conceitos trabalhados nos capítulos que compõem esta parte do curso:

- **Capítulo 1 – Fundamentos de computação**. Introduz noções básicas sobre como computadores funcionam, o que significa programar e quais as características que diferenciam as várias linguagens de programação disponíveis. O objetivo é construir uma base conceitual mínima para entender o que acontece "por trás" do código.
- **Capítulo 2 – Primeiros passos no Python**. Apresenta o Python como linguagem de programação, discutindo sua sintaxe básica, a execução de comandos simples e a interação com o ambiente de desenvolvimento. Este capítulo marca o primeiro contato prático com a escrita e execução de código.
- **Capítulo 3 – Tipos de dados e objetos básicos**. Introduz os principais tipos de dados do Python – como números, *strings* e booleanos – e os objetos básicos utilizados para armazenar e manipular os diferentes tipos de informação. Este capítulo estabelece a base para compreender como a informação é representada em programas.
- **Capítulo 4 – Controle de fluxo e iteração**. Apresenta estruturas fundamentais para controlar a execução de um programa, como condicionais e laços de repetição (*loops*, em inglês). São discutidas formas de automatizar tarefas repetitivas e de implementar lógica condicional, elementos centrais em qualquer aplicação computacional.
- **Capítulo 5 – Funções**. Este capítulo tem por objetivo desenvolver o conceito de funções como forma de organizar código, evitar repetição e tornar programas mais legíveis e reutilizáveis. Discute definição, uso e boas práticas na criação de funções, preparando o aluno para estruturar programas mais complexos.
- **Capítulo 6 – Exceções e Expressões Regulares**. Este capítulo apresenta os tipos de erros mais comuns que podem ocorrer durante a execução de um programa em Python. O objetivo é capacitar o leitor a identificar e distinguir cada tipo de erro, facilitando o processo de depuração e tornando o desenvolvimento de código mais eficiente e seguro. Na segunda parte, o capítulo introduz as expressões regulares ("regular expressions" ou "regex"), uma ferramenta poderosa para identificar e manipular padrões em textos. As expressões regulares permitem analisar e validar informações de forma precisa, ampliando as possibilidades de tratamento e verificação de dados dentro dos programas.

Juntos, esses seis capítulos estabelecem as bases conceituais e práticas da programação, permitindo que o aluno desenvolva familiaridade com o raciocínio computacional e com a linguagem Python. Ao final desta parte do curso, mais do que memorizar comandos específicos, espera-se que o estudante seja capaz de ler, compreender e construir pequenos programas, criando uma base sólida para o uso de ferramentas computacionais em análise de dados e aplicações empíricas em economia nas etapas seguintes da disciplina.

1 Como um computador funciona

1.1 Introdução

Antes de escrever qualquer linha de código, é importante entender o ambiente no qual a programação acontece. Programar não é apenas aprender uma linguagem específica, mas também compreender como o computador organiza arquivos, executa comandos e interage com o sistema operacional. Muitos dos erros e dificuldades enfrentados por iniciantes em programação não estão relacionados à lógica do código em si, mas ao uso do computador como ferramenta de trabalho: navegar por diretórios, localizar arquivos, executar programas e interpretar mensagens do sistema. Esses aspectos são frequentemente tratados como triviais, mas são fundamentais para o uso eficiente de qualquer linguagem de programação.

Este capítulo introduz conceitos fundamentais sobre o funcionamento de computadores, a representação de informações em bits e bytes, a organização de arquivos e a forma como interagimos com o sistema operacional via terminal. Esses elementos servirão como infraestrutura conceitual para todo o restante do curso. O foco aqui não é aprofundar detalhes técnicos de hardware ou sistemas operacionais, mas fornecer uma visão prática e funcional do ambiente computacional que será utilizado ao longo da disciplina.

1.1.1 Pré-requisitos

Neste capítulo introdutório não utilizaremos nenhum programa ou linguagem. Ele foi escrito para que mesmo o aluno sem nenhum tipo de conhecimento em computação e com o mínimo de conhecimento a respeito de operações matemática elementares tenha capacidade de acompanhar. Basta curiosidade!

1.2 Como um computador funciona

Um computador realiza duas coisas e apenas duas coisas: faz operações e guarda de forma eficiente o resultado dessas operações. No entanto, ao fazer essas duas coisas de forma extremamente eficiente essa máquina chamada computador nos permite receber, armazenar, processar e transmitir informações. Em essência, o computador nos serve ao propósito de **resolução de problemas**. Isso significa que, através de uma linguagem formal (especificamente operações de computação), podemos formular problemas e programar meios de resolver de

Figura 1.1: Setup usual de um computador pessoal no começo dos anos 2000. Fonte: *Wikimedia Commons*

forma automatizada tais problemas. No fim das contas, programar é o ato de criar programas, isto é, estabelecer uma sequência de instruções em linguagem formal que especifica como executar uma determinada operação no computador.

Mas antes de avançar na ideia de programa e algoritmos, quais são as características de um computador como o conhecemos?

1.2.1 Sistema computacional

Um sistema computacional é resultado da integração de componentes atuando como uma entidade, com o propósito de processar dados, isto é, realizar algum tipo de operação lógica envolvendo os dados, de modo a produzir diferentes níveis de informações. Tais sistemas são compostos por 3 principais elementos: hardware, software e o componente humano. O hardware inclui os componentes físicos enquanto o software é o conjunto de instruções que diz ao hardware o que fazer. No entanto, o software não elabora e decide sozinho o tipo de instrução a ser enviada ao hardware, é só através do componente humano que as instruções e objetivos são definidos.

- **Hardware**: componente físico de um sistema de computação, isto é, todos os equipamentos utilizados pelo usuário nas ações de entrada, processamento, armazenamento e saída de dados. Exemplos:
  - Teclado, mouse, impressora
  - Unidade central de processamento (CPU, em inglês) ou processador
  - Placa mãe
  - Placa de vídeo (GPU)
  - Memória RAM
  - Disco rígido de armazenamento
- **Software**: componente lógico de um sistema de computação, isto é, séries de instruções que fazem o computador funcionar. São os famosos programas de computador, que podem ser (i) proprietários e utilizáveis a partir da compra de uma licença ou (ii) gratuitos. Exemplos:
  - Editores de texto (e.g., Microsoft Word)
  - Editor de planilhas (e.g., Microsoft Excel)
  - Browsers para navegação na internet (e.g., Safari e Google Chrome)
- **Componente humano**: as pessoas, que utilizam o computador basicamente como ferramenta para atingir determinado fim, para resolver problemas. Sem a ação de um indivíduo o sistema computacional não funciona, já que (ainda) depende das instruções e direcionamento humano para definir o que o software deve fazer e como deve interagir com o hardware.

1.2.2 Arquitetura de um computador

A arquitetura de um computador define como os componentes estão interconectados e como eles colaboram para executar tarefas. Os computadores, como os conhecemos hoje, são estruturados em cima da lógica proposta inicialmente por John Von Neumann, matemático húngaro que viveu durante a primeira metade do século XX e que contribuiu imensamente em várias áreas das ciências. Embora tenhamos hoje uma complexidade maior nos elementos que compõe a arquitetura de um computador, a estrutura básica por trás dos computadores mais modernos continua sendo aquela proposta por Von Neumann. Essa arquitetura pode ser representado pelo diagrama abaixo:

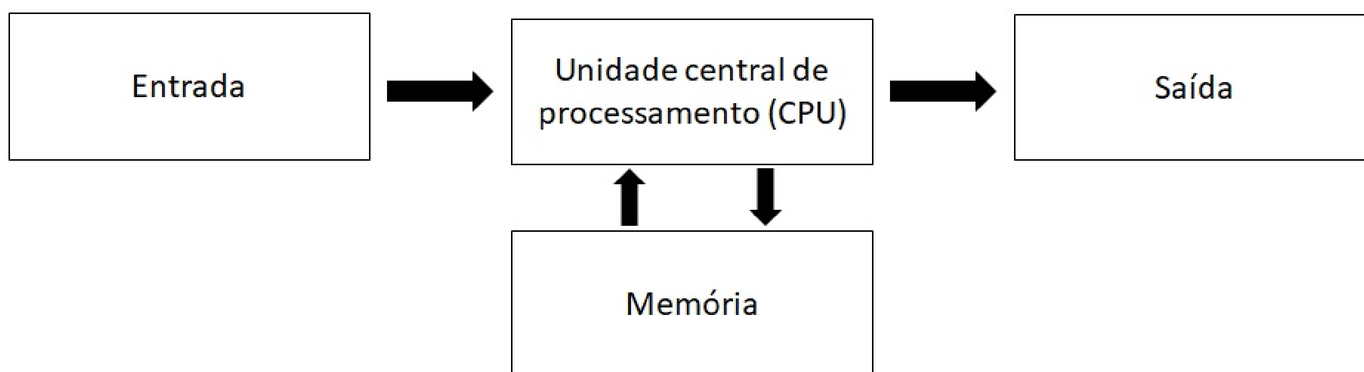


Figura 1.2: Diagrama que representa o funcionamento da arquitetura Von-Neumann

Essencialmente, o hadware que compõe as partes da arquitetura de um computador podem ser dividas em 3 partes:

- **Dispositivos de entrada e saída**: é através deles que o usuário dialoga com a máquina. Através do teclado e mouse, por exemplo, o usuário fornece informações ao computador a partir das quais processos serão realizados e seus resultados serão percebidos pelos dispositivos de saída (monitor e impressora, por exemplo).
- **Processador ou CPU**: é o cérebro do sistema computacional, é nele que serão executados cálculos e instruções lógicas dos usuários.
- **Memória**: aqui são armazenados temporariamente os resultados dos cálculos, dados e instruções. Esse armazenamento na memória, porém, não é feito de forma permanente, ele serve apenas como suporte ao processador. É como se o processador utilizasse a memória de caderno de anotações, retomando o resultado dos cálculos já realizados sempre que necessário.

! Importante

Não confunda *memória* com o que chamamos de *armazenamento*. A memória do computador, normalmente na forma da chamada memória RAM (sigla para *Random Access Memory*), é responsável por armazenar temporariamente os dados de dos programas que estão em uso, permitindo acesso rápido pelo processador. É também chamada de memória volátil: todo o seu conteúdo é perdido quando o computador é desligado.
Já as informações armazenadas no disco rígido (*Hard Disk Drive*, HDD) ou na unidade de estado sólido (*Solid-State Drive*, SSD) não são perdidas quando o computador é desligado. Esses dispositivos mantêm os dados armazenados de forma persistente, funcionando como a "memória de longo prazo" do computador. Por isso, esse tipo de armazenamento é chamado de memória não volátil.
Fazendo uma analogia com o funcionamento do nosso cérebro, a memória RAM armazena informações de curto prazo, como o que você almoçou hoje. Esse tipo de informação é importante ao longo do dia, por exemplo para saber se você já se alimentou bem ou se ainda está longe da sua meta diária de proteína, mas costuma ser esquecida depois que você dorme.
Já o HDD ou SSD armazena informações de longo prazo, como memórias da infância ou experiências passadas. Essas informações não são utilizadas constantemente no dia a dia, mas permanecem disponíveis e podem ser úteis em conversas com amigos ou como aprendizado acumulado, por exemplo na hora de decidir como cuidar dos seus filhos quando chegar a hora.

1.2.3 Bits e bytes

Para entender como um computador armazena e processa informações, é necessário compreender o conceito de sistema binário ou sistema de base 2[1]. Diferentemente do sistema decimal, que utilizamos no dia a dia e possui dez dígitos (de 0 a 9), os computadores operam utilizando apenas dois estados possíveis: 0 e 1. Pense nisso como um interruptor, em que 1 representa o estado "ligado" e 0 o estado "desligado". Os processadores atuais de um computador são resultado da junção de bilhões desses minúsculos "interruptores", também chamados de transistores.

Esses dois estados, 0 ou 1, também são chamados de bits (do inglês *binary digits*). Um bit representa a menor unidade de informação que um computador pode armazenar. Qualquer informação processada por um computador – números, textos, imagens ou vídeos – é, em última instância, representada como uma sequência de bits.

Como trabalhar com bits individualmente seria pouco prático, eles são agrupados em unidades maiores chamadas de bytes. Um byte é composto por 8 bits e permite representar um conjunto maior de valores. Um byte pode representar 256 ($2^8 = 256$) combinações diferentes, o que é

[1]Para entender um pouco mais sobre a história do sistema binário acesse o link.

suficiente para representar todas as letras maiúsculas e minúsculas, números e símbolos básicos do alfabeto inglês. Não à toa, o byte se tornou a unidade padrão de medida quando o assunto é capacidade de armazenar informação em um computador. A Tabela Tabela 1.1 mostra como converter alguns números do sistema decimal para o sistema binário de 8 bits, em que cada dígito dibário indica a presença ou ausência de uma potência de 2.

Tabela 1.1: Representação de números decimais no sistema binário e detalhamento do cálculo para conversão.

Número decimal	Número binário	Cálculo para conversão
0	00000000	$0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+0\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0=0$
1	00000001	$0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+0\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0=1$
5	00000101	$0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+0\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0=5$
48	00110000	$0\cdot2^7+0\cdot2^6+1\cdot2^5+1\cdot2^4+0\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0=48$
137	10001001	$1\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+1\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0=137$
255	11111111	$1\cdot2^7+1\cdot2^6+1\cdot2^5+1\cdot2^4+1\cdot2^3+1\cdot2^2+1\cdot2^1+1\cdot2^0=255$

A partir dos bytes, surgem unidades maiores, amplamente utilizadas para medir o tamanho de arquivos, a quantidade de memória e a capacidade de armazenamento: quilobytes, megabytes, gigabytes, terabytes e petabytes. Essas unidades indicam, essencialmente, quantos bytes são necessários para representar uma determinada quantidade de informação.

- Kilobyte (KB): 1,024 (ou 2^{10}) bytes.
- Megabyte (MB): 1,048,576 (ou 2^{20}) bytes.
- Gigabyte (GB): 1,073,741,824 (ou 2^{30}) bytes.
- Terabyte (TB): 1,099,511,627,776 (ou 2^{40}) bytes.
- Petabyte (PB): 1,125,899,906,842,624 (ou 2^{50}) bytes.
- Exabyte, Zettabyte, Yottabyte…

Compreender bits e bytes é fundamental para entender por que arquivos têm tamanhos diferentes, por que algumas operações exigem mais memória do que outras e por que certas tarefas computacionais são mais custosas do ponto de vista de processamento. Esses agrupamentos de bits (bytes) são as unidades básicas que a memória e o armazenamento usam para guardar informação. A próxima seção explica o papel desses componentes e como eles impactam o desempenho prático.

1.2.4 ASCII

Assim como os números são representados por padrões binários de zeros e uns, os caracteres de texto — letras, números, sinais de pontuação e símbolos — também são codificados como sequências de bits. Como os mesmos padrões binários poderiam ser interpretados de maneiras distintas, tornou-se necessário estabelecer uma convenção padronizada.

Uma das primeiras e mais importantes convenções foi o ASCII (*American Standard Code for Information Interchange*). O ASCII associa um número inteiro específico a cada caractere, permitindo que computadores representem texto de forma consistente. Por exemplo, a letra `A` está associada ao número decimal 65, que em binário é representado por `01000001`.

Suponha que você recebesse uma sequência binária correspondente aos números decimais 72, 73 e 33. Ao convertê-los segundo a tabela ASCII, obteríamos os caracteres `H, I e !`, formando a expressão `HI!`. É graças a padrões como o ASCII que diferentes computadores "concordam" sobre qual número representa qual caractere.

A tabela a seguir apresenta um mapa dos valores ASCII:

Tabela 1.2: Tabela ASCII.

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	”	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	ˆ	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Note que cada caractere ASCII ocupa 1 byte (8 bits). Com 8 bits, é possível representar até 256 valores distintos (de 0 a 255). No entanto, o ASCII padrão utiliza apenas os valores de 0 a 127, totalizando 128 caracteres. Os códigos de 0 a 31 e o 127 correspondem a caracteres de controle (como quebra de linha e tabulação), enquanto os valores de 32 a 126 representam caracteres imprimíveis.

1.2.5 UNICODE

Com a expansão da comunicação digital em escala global, o limite de 128 caracteres do ASCII original - e mesmo as extensões de 256 caracteres - tornou-se insuficiente. Era necessário um

padrão capaz de representar diferentes alfabetos, sistemas de escrita, símbolos matemáticos, caracteres técnicos e, mais recentemente, emojis.

O Unicode surgiu com esse objetivo: criar um sistema universal de codificação capaz de representar praticamente todos os sistemas de escrita utilizados pela humanidade, além de símbolos e elementos gráficos. Diferentemente do ASCII, o Unicode não está limitado a 8 bits. Ele define um conjunto muito mais amplo de códigos (chamados code points), que podem ser representados por diferentes esquemas de codificação, como UTF-8, UTF-16 e UTF-32. O UTF-8, por exemplo, é hoje o padrão dominante na internet.

Embora o código numérico associado a cada caractere seja universal, sua aparência visual pode variar entre sistemas e fabricantes. Assim, um mesmo emoji pode ter pequenas diferenças de design em dispositivos da Apple, Google ou Microsoft. O padrão Unicode continua em expansão, incorporando novos caracteres e emojis para refletir transformações culturais, linguísticas e tecnológicas.[2]

1.2.6 RGB

Os mesmos zeros e uns que representam letras e números também podem representar cores. No sistema RGB (Red, Green, Blue), cada cor é definida pela combinação das intensidades de vermelho, verde e azul. Cada uma dessas três intensidades é representada por um número entre 0 e 255 — ou seja, 8 bits para cada canal de cor. Assim, uma cor no sistema RGB utiliza 3 bytes (24 bits no total):

Por exemplo, a sequência numérica (72, 73, 33) — que no contexto ASCII corresponde a `HI!` — poderia, em um contexto de imagem, ser interpretada como uma cor específica. Essa combinação resulta em um tom escuro amarelado-esverdeado:

Cada conjunto de três valores RGB forma um pixel (um ponto de cor). Uma imagem digital nada mais é do que uma grande matriz de pixels, isto é, uma matriz de números. Quanto maior

[2] Se desejar se aprofundar no tema, você pode consultar mais informações sobre o sistema UNICODE e sobre emojis.

o número de pixels, maior a resolução da imagem. Da mesma forma, sons e vídeos também são representados por sequências numéricas. No caso do áudio, os números codificam variações de amplitude ao longo do tempo; no caso do vídeo, combinam-se sequências de imagens (frames) com informação sonora. Em última instância, todo conteúdo digital — texto, imagem, som ou vídeo — é armazenado como padrões de zeros e uns.

1.2.7 Processamento, memória e armazenamento na prática

Podemos entender o funcionamento de um computador como um fluxo contínuo de **leitura**, **processamento** e **armazenamento** de bits. Quando observamos as especificações de um computador pessoal, é comum encontrar informações como número de núcleos do processador (*cores*), frequência do processador em gigahertz (GHz), quantidade de memória RAM e capacidade de armazenamento em disco. Esses números descrevem, essencialmente, a velocidade e a escala com que o computador consegue manipular grandes volumes de informação binária em cada etapa desse fluxo contínuo.

Processamento refere-se à execução de instruções sobre dados representados em bits. O processador é o componente responsável por realizar operações elementares que, em última instância, são operações lógicas sobre sequências de zeros e uns. A frequência do processador indica aproximadamente quantos ciclos de processamento podem ser realizados por segundo. Frequências mais altas permitem que mais instruções sejam executadas em menos tempo, embora o desempenho final dependa também da organização do código, do tipo de tarefa e do acesso à memória. Além disso, processadores modernos possuem múltiplos *cores* ou núcleos. O uso de múltiplos núcleos permite paralelismo, isto é, a execução simultânea de diferentes partes de uma tarefa, de forma análoga a dividir um trabalho entre várias pessoas para concluí-lo mais rapidamente.

Memória (RAM) é o espaço onde os dados e instruções ficam armazenados temporariamente enquanto estão sendo processados. É na memória RAM que o computador mantém os bytes que serão acessados repetidamente pelo processador. Quanto maior a quantidade de RAM disponível, maior é o volume de dados que pode ser mantido "em uso" ao mesmo tempo. Quando a RAM é insuficiente, o computador passa a usar o disco como apoio, o que reduz drasticamente o desempenho.

Armazenamento, por sua vez, diz respeito ao local onde os dados são guardados de forma permanente. HDDs e SSDs armazenam grandes volumes de bytes mesmo quando o computador está desligado. A diferença principal é que SSDs permitem acesso muito mais rápido aos dados, reduzindo o tempo necessário para carregar programas, abrir arquivos e transferir grandes conjuntos de dados para a memória.

Vistos em conjunto, processamento, memória e armazenamento formam um sistema integrado: dados são lidos do disco, carregados na memória, processados pela CPU e, quando necessário, gravados novamente no armazenamento. Para tarefas simples, esse fluxo passa despercebido.

Para programação, análise de dados e computação numérica compreender essas etapas ajuda a interpretar erros, gargalos de desempenho e limitações práticas do ambiente computacional.

O que significam as especificações do seu computador?

Considere um computador com processador com 16 núcleos e frequência de 3.2 GHz, 32 GB de memória RAM e 1 TB de SSD. Esses números descrevem como ele lida com bits e bytes em diferentes etapas do processamento.

- **16 núcleos (cores)** indicam que o processador pode executar várias tarefas ao mesmo tempo. Em aplicações que permitem paralelismo – como certas operações numéricas ou análises sobre grandes conjuntos de dados – isso é semelhante a dividir o trabalho entre dezesseis pessoas em vez de apenas uma.
- **3.2 GHz** refere-se à velocidade com que cada núcleo executa instruções elementares. Em termos simples, indica quantos passos o processador consegue dar por segundo ao manipular dados representados em sistema binário.
- **32 GB de RAM** significam que até esse volume de dados pode permanecer disponível para acesso rápido enquanto programas estão em execução. Bases de dados grandes, matrizes extensas e múltiplos processos consomem memória rapidamente.
- **1 TB de SSD** indica a capacidade de armazenamento permanente.

Em análise de dados e programação, problemas de desempenho raramente estão ligados apenas à velocidade do processador. Com frequência, eles surgem porque os dados são grandes demais para caber na memória ou porque operações são realizadas de forma pouco eficiente. Entender essas especificações ajuda a interpretar erros, lentidão e limitações práticas do ambiente computacional.

1.3 Arquivos e diretórios

Mais do que armazenar informação, é importante entender *onde* e *como* a informação é armazenada. Assim como na vida cotidiana, é preciso ter organização no armazenamento para saber como e onde procurar determinada informação no computador. Se você é uma pessoa organizada, você sabe que o iogurte estará sempre guardada na geladeira e em algum pote fechado, não é preciso procurar debaixo do chuveiro em um saco de pão.

1.3.1 Estrutura hierárquica

Os diretórios servem justamente para organizar o armazenamento do computador. É como se você pudesse separar o seu SSD primeiro em cômodos, depois em armários dentro de cada cômodo, em gavetas dentro de cada armário e assim por diante. Formalmente, um diretório pode ser entendido como um contêiner no qual arquivos e outros diretórios podem estar organizados, formando uma estrutura hierárquica. Essa organização permite localizar, acessar e manipular arquivos de forma sistemática, especialmente em projetos maiores.

Por exemplo, a organização de um projeto de análise de dados pode ter a seguinte forma:

```
C:\
  projeto\
      data\
          dados.csv
      scripts\
          analise.py
      resultados\
          figura1.png
```

Nesse exemplo, `projeto` é um diretório que está dentro da partição C e que contém outros diretórios. O arquivo `dados.csv`, por exemplo, está dentro do diretório `data`, que é um diretório localizado dentro do diretório `projeto`. Para acessar esse arquivo `dados.csv` dentro dessa estrutura hierárquica de diretórios utilizamos duas formas principais: o caminho absoluto ou o caminho relativo.

- **Caminho absoluto**: descreve a localização completa a partir da raiz do sistema de arquivos. Nesse caso, o caminho absoluto seria `C:\projeto\data\dados.csv`.
- **Caminho relativo**: descreve a localização em relação ao diretório atual. Isso quer dizer que, se estivermos trabalhando em um programa a partir do diretório `projeto`, para acessar o arquivo de dados basta descrever o caminho relativo a partir dali: `data/dados.csv`.

1.3.2 Extensões de arquivos

Para além do *onde*, a informação em si pode ser armazenada de várias formas. Você pode guardar o iogurte em uma jarra de vidro ou mantê-lo no recipiente original. Um conjunto de informação é armazenada em um arquivo que pode ter vários formatos ou **extensões**. A extensão do arquivo corresponde ao sufixo após o ponto no nome do arquivo e indica como os dados estão organizados internamente e quais programas sabem interpretá-los corretamente. Do ponto de vista conceitual, a extensão não “muda” o dado em si, mas define a estrutura lógica usada para representar a informação. O mesmo conjunto de informações pode ser salvo

em formatos diferentes, dependendo do objetivo (leitura humana, processamento automático, troca entre sistemas etc.).

Um formato muito comum em análise de dados é o CSV (`.csv`, *Comma-Separated Values*). Trata-se de um arquivo de texto em que os dados são organizados em linhas e colunas, sendo as colunas separadas por um delimitador, geralmente uma vírgula. Cada linha representa uma observação e a primeira linha costuma conter os nomes das variáveis. Arquivos CSV são amplamente utilizados porque (i) são simples e legíveis; (ii) podem ser abertos em editores de texto, planilhas e softwares estatísticos; e (iii) são facilmente importados por linguagens como Python.

Outro formato bastante utilizado é o JSON (`.json`, *JavaScript Object Notation*). Embora também seja um arquivo de texto, sua estrutura é diferente. Em vez de linhas e colunas, os dados são organizados como pares chave–valor, de forma semelhante a um dicionário, podendo conter listas e estruturas aninhadas. Esse formato é muito usado para troca de dados entre sistemas e aplicações, além do uso para armazenamento de configurações.

Outras extensões comuns incluem:

- `.txt`: texto simples, sem estrutura fixa.
- `.xlsx`: arquivos de planilha do Microsoft Excel.
- `.py`: arquivos de código Python.

É importante enfatizar que a extensão não garante o conteúdo do arquivo: um `.csv` mal formatado pode não ser lido corretamente, e um arquivo `.txt` pode conter dados altamente estruturados. Ainda assim, seguir convenções de extensão é essencial para organização de projetos, reprodutibilidade e interoperabilidade entre ferramentas. Considere a tabela de notas abaixo:

nome	nusp	ingresso	materia	media
Joao	12345678	2026	EAE1106	8
Miguel	5253678	2023	EAE1106	4
Alice	9743678	2025	EAE1106	9

O armazenamento desses dados em formato CSV e JSON é feito da seguinte maneira:

- Formato CSV

```
nome,nusp,ingresso,materia,media
Joao,12345678,2026,EAE1106,8
Miguel,52543678,2023,EAE1106,4
Alice,9743678,2025,EAE1106,9
```

- Formato JSON

```
[
{
   "nome": "Joao",
   "nusp": 12345678,
   "ingresso": 2026,
   "materia": "EAE1106",
   "media": 8
},
{
   "nome": "Miguel",
   "nusp": 52543678,
   "ingresso": 2023,
   "materia": "EAE1106",
   "media": 4
},
{
   "nome": "Alice",
   "nusp": 9743678,
   "ingresso": 2025,
   "materia": "EAE1106",
   "media": "9"
}
]
```

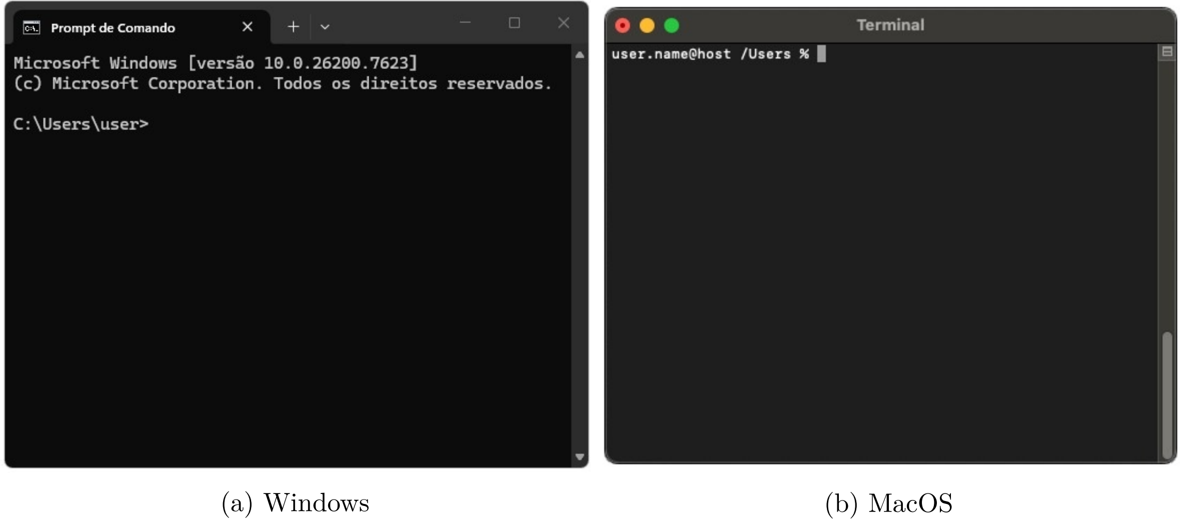
Mas qual a importância disso tudo? Entender como os computadores armazenam números e como arquivos e diretórios são organizados e acessados pelo sistema operacional ajuda a compreender limitações e comportamentos observados na prática. Por exemplo, explica por que um sistema de 32 bits é capaz de alocar apenas até 4 GB de memória RAM ou por que arquivos com milhões de linhas podem ser difíceis de abrir ou processar em um computador local.

Ter um conhecimento mínimo sobre o funcionamento do computador e sobre como os programas interagem com sua arquitetura permite entender o que ocorre "por trás das cortinas". Com isso, podemos usar o computador como uma ferramenta para resolver problemas — e não como um fim em si. Para chegar a esse ponto, precisamos de meios formais para nos comunicar com o sistema operacional e, posteriormente, expressar instruções de forma lógica e precisa.

1.4 Terminal

Um passo intermediário entre o funcionamento interno do computador e as linguagens de programação é o terminal. O terminal é uma interface textual que permite interagir diretamente com o sistema operacional por meio de comandos simples. Em vez de clicar em ícones ou navegar por menus gráficos, o usuário descreve explicitamente o que deseja que o computador faça.

O uso do terminal torna visíveis conceitos introduzidos na seção anterior, como arquivos e diretórios, além de fornecer um primeiro contato com a lógica de comandos que será essencial ao longo do curso. A Figura Figura 1.3 ilustra o ambiente típico do terminal nos dois principais sistemas operacionais em uso atualmente.



(a) Windows (b) MacOS

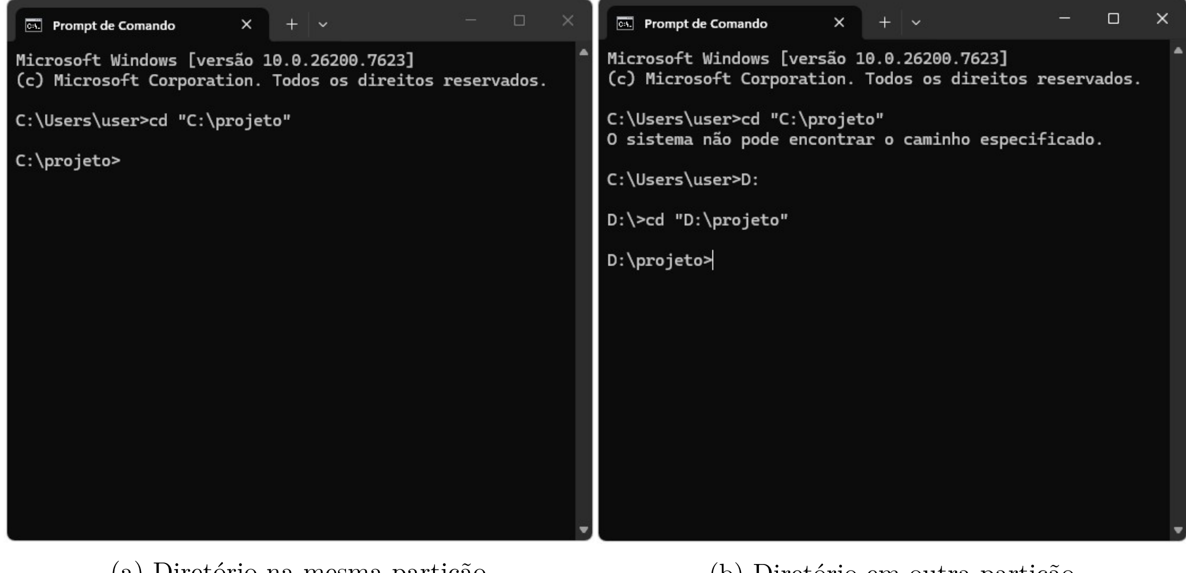
Figura 1.3: Interface usual do terminal

Comandos básicos no terminal (ou prompt de comando) do Windows incluem:

- `dir`: lista arquivos e diretórios no ambiente atual
- `cd`: muda de diretório
- `mkdir`: cria um novo diretório

No exemplo do painel A da Figura Figura 1.3, `cd "C:\projeto"` muda o ambiente atual do diretório `C:\Users\user` para `C:\projeto`. Note, porém, que se a pasta `projeto` estiver em outra partição do seu armazenamento, é preciso primeiro mudar para essa partição para depois utilizar o comando `cd`. A Figura Figura 1.4 ilustra esses dois casos.

Mesmo que você utilize interfaces gráficas, compreender o terminal facilita o entendimento de como o computador está organizado e de como o sistema operacional executa tarefas básicas.



(a) Diretório na mesma partição (b) Diretório em outra partição

Figura 1.4: Exemplo de uso do terminal para mudar o ambiente atual

Ao longo do curso, o terminal será utilizado principalmente para executar programas, navegar entre diretórios e instalar novos pacotes e ferramentas.

Apesar de útil, o terminal ainda exige que as instruções sejam dadas comando a comando, de forma relativamente limitada e pouco reutilizável. Se queremos resolver problemas mais gerais e automatizar tarefas de forma sistemática, é necessário um nível adicional de abstração.

1.5 Linguagens de programação

Para atingir esse nível adicional de abstração, precisamos de um meio de comunicar instruções ao computador de maneira precisa e sem ambiguidades. Esse meio são as linguagens de programação, que definem regras formais para descrever operações, decisões e cálculos que o computador pode executar. Diferentemente das linguagens naturais, como o português que eu e você falamos, as linguagens de programação são linguagens formais, que foram criadas pelas pessoas para aplicações específicas. A notação usada pelos matemáticos é uma linguagem formal adequada para representar relações entre números e símbolos; químicos utilizam linguagens formais para representar a estrutura de moléculas. Utilizamos as linguagens de programação para nos comunicar de forma inequívoca com os computadores.

1.5.1 Sintaxe e semântica

Toda linguagem de programação possui dois níveis fundamentais: sintaxe e semântica.

A **sintaxe** refere-se às regras de escrita da linguagem. Ela define quais símbolos e quais sequências de símbolos são válidas: onde usar parênteses, palavras-chave, operadores e a ordem correta dos elementos. Erros de sintaxe são, em geral, fáceis de identificar porque a execução do programa é interrompida e uma mensagem de erro é apresentada. Exemplos comuns incluem esquecer um caractere obrigatório em determinada sequência, utilizar parênteses, colchetes ou aspas sem o fechamento correspondente ou escrever palavras-chave de forma incorreta. Nesses casos, o computador não consegue sequer interpretar a estrutura do código, independentemente da intenção do programador.

Por outro lado, a **semântica** refere-se ao significado das instruções. Um código pode estar sintaticamente correto, mas produzir resultados errados por não representar corretamente a intenção do programador. Erros semânticos são mais sutis, pois o código respeita todas as regras formais da linguagem e é executado normalmente, mas o resultado obtido está incorreto. Por exemplo, um programa pode calcular uma média dividindo a soma dos valores pelo número errado de observações ou usar uma variável diferente da pretendida em uma expressão. Esses erros não geram mensagens automáticas do computador e só podem ser detectados pela análise crítica dos resultados produzidos.

Essa distinção é central em programação. O computador consegue verificar automaticamente erros de sintaxe, mas não é capaz de julgar se o significado do programa está correto do ponto de vista do problema que se deseja resolver.

1.5.2 Diferenças entre linguagens naturais e formais

Embora as linguagens formais e naturais tenham muitas características em comum – símbolos, estrutura, sintaxe e semântica – há algumas diferenças:

1. **Ambiguidade**: as linguagens naturais são cheias de ambiguidade e as pessoas lidam com isso usando pistas contextuais e outras informações. As linguagens formais são criadas para ser quase ou completamente inequívocas, ou seja, qualquer afirmação tem exatamente um significado, independentemente do contexto
2. **Redundância**: para compensar a ambiguidade e reduzir equívocos, as linguagens naturais usam muita redundância. Por causa disso, muitas vezes são verborrágicas. As linguagens formais são menos redundantes e mais concisas.
3. **Literalidade**: as linguagens naturais são cheias de expressões e metáforas. Se eu digo "caiu a ficha", provavelmente não há ficha nenhuma na história, nem nada que tenha caído (esta é uma expressão para dizer que alguém entendeu algo depois de certo período de confusão). As linguagens formais têm significados exatamente iguais ao que expressam.

Como todos nós crescemos falando linguagens naturais, às vezes é difícil se ajustar a linguagens formais. As linguagens formais são mais densas que as naturais, então exigem mais tempo para a leitura. Além disso, a estrutura é importante, então nem sempre é melhor ler de cima para baixo e da esquerda para a direita. Em vez disso, aprenda a analisar o programa primeiro, identificando os símbolos e interpretando a estrutura. E os detalhes fazem diferença. Pequenos erros em ortografia e pontuação, que podem não importar tanto nas linguagens naturais, podem fazer uma grande diferença em uma língua formal.

1.5.3 Alto nível X baixo nível

As linguagens de programação também diferem quanto ao seu nível de abstração. Na ciência da computação, **linguagens de programação de alto nível** abstraem detalhes do funcionamento interno do computador, permitindo que o programador se concentre na lógica do problema. São linguagens mais próximas das linguagens humanas e mais distantes das linguagens de máquina (e do sistema binário). Tais linguagens podem usar elementos de linguagem natural, serem mais fáceis de usar, ou podem ocultar totalmente áreas significativas de sistemas de computação, como o gerenciamento de memória. Isso torna o processo de desenvolvimento de um programa mais simples e eficiente. Python, R e Java são alguns exemplos de linguagens de alto nível.

Por outro lado, uma **linguagem de baixo nível** é uma linguagem de programação que fornece pouca ou nenhuma abstração de conceitos de programação e está muito próxima de escrever instruções de máquina reais. A palavra "baixo" refere-se à pequena ou inexistente quantidade de abstração entre a linguagem e a linguagem de máquina; por causa disso, as linguagens de baixo nível são às vezes descritas como *próximas do hardware*. Programas escritos em linguagens de baixo nível tendem a ser relativamente não portáteis e dependentes do computador para o qual foram escritas. Assembly é uma das linguagens de mais baixo nível à disposição do programador.

Em cursos como Economia e outras ciências aplicadas, linguagens de alto nível são preferidas porque reduzem o custo cognitivo, o tempo necessário para produzir código e aumentam a produtividade.

1.5.4 Linguagem Compilada X Interpretada

Outra distinção importante diz respeito à forma como o código é executado. Em **linguagens compiladas** o código é traduzido integralmente para código de máquina antes da execução. Elas também dão ao desenvolvedor mais controle sobre os aspectos de hardware, como gerenciamento de memória e uso da CPU. As linguagens compiladas precisam de uma etapa de "construção" – elas precisam ser compiladas manualmente primeiro. Somente após essa etapa que o código pode rodar por completo. Linguagens como C e C++ seguem esse modelo.

Em **linguagens interpretadas**, as instruções não são executadas diretamente pela máquina de destino, mas lidas e executadas, linha por linha, por algum outro programa, um intérprete. Isso permite maior flexibilidade e facilita a experimentação, embora geralmente com menor desempenho em tempo de máquina. Python e R são exemplos clássicos de linguagens interpretadas.

Embora usualmente mais lentas, essa simplicidade e flexibilidade em escrever código torna linguagens interpretadas particularmente adequadas para aprendizado, análise de dados e trabalho empírico. Lembre-se, no fim das contas, o tempo relevante não é apenas o tempo de máquina, mas o tempo que você leva para escrever e depurar código.

1.5.5 Paradigmas

Linguagens de programação também podem ser classificadas segundo o paradigma que adotam, isto é, o estilo predominante de organização do código. Entre os principais paradigmas de programação podemos citar programação imperativa, procedural, funcional, declarativa e programação orientada a objetos. Na programação imperativa, por exemplo, o programador diz como, o quê e em qual ordem exatamente um programa ou rotina deve realizar. É neste paradigma que surgiram as famosas estruturas condicionais e atribuição de valor à variáveis. Por outro lado, na programação orientada a objetos o programa é estruturado em objetos que combinam dados e comportamento. Mais detalhes sobre paradigmas de programação podem ser obtidos aqui.

Muitas linguagens modernas, incluindo Python, são multiparadigma, permitindo combinar diferentes estilos conforme o problema.

1.5.6 Tipagem

Por fim, as linguagens diferem quanto à forma como tratam os tipos de dados. Em linguagens de tipagem estática, o tipo de cada variável é definido explicitamente no código e verificado antes da execução. O Python utiliza tipagem dinâmica, isso significa que o próprio interpretador do Python infere o tipo dos dados que uma variável recebe, sem a necessidade que o usuário diga de que tipo determinada variável é. Além disso, o Python é uma linguagem fortemente tipada, o que significa que o interpretador do Python avalia as expressões por conta própria e não faz conversões automáticas de valores entre tipos de dados não compatíveis. Ao fazer operações com tipos incompatíveis, o Python não converte automaticamente esses tipos pra você, ele vai dar erro.

Tipagem dinâmica reduz a quantidade de código inicial e facilita experimentação, mas exige atenção adicional para evitar erros lógicos já que o usuário não define explicitamente o tipo do dado. No entanto, a tipagem forte reduz a chance de resultados inesperados por conta de tipos definidos ou convertidos de forma equivocada já que a linguagem não tenta corrigir o erro por conta própria.

1.6 Algoritmos

A resolução de problemas está no centro da Ciência da Computação. Um algoritmo é um conjunto finito e bem definido de instruções, organizadas passo a passo, para resolver um problema ou realizar uma tarefa. Considere um problema simples: encontrar um nome em uma lista telefônica. Como poderíamos resolvê-lo?

Uma abordagem seria percorrer a lista página por página, do início ao fim, até localizar o nome desejado. Outra possibilidade seria examinar duas páginas por vez, tentando acelerar o processo. Uma estratégia ainda mais eficiente seria abrir a lista aproximadamente no meio e perguntar: "O nome que procuro está antes ou depois desta página?". A partir dessa resposta, descarta-se metade da lista e repete-se o procedimento na parte restante, reduzindo o problema pela metade a cada etapa.

Cada uma dessas estratégias constitui um algoritmo - isto é, uma sequência organizada de instruções para alcançar um objetivo. Programadores transformam esse tipo de raciocínio em código, permitindo que o computador execute o algoritmo de forma automática e precisa.

1.7 Pseudocódigo

O pseudocódigo é uma forma de descrever algoritmos utilizando uma linguagem próxima da linguagem natural, mas estruturada de maneira semelhante a um programa de computador. Ele não segue rigorosamente a sintaxe de uma linguagem específica, mas organiza o raciocínio lógico de forma clara e sequencial.

Considere, por exemplo, o terceiro algoritmo utilizado para resolver o problema da lista telefônica. Poderíamos descrevê-lo em pseudocódigo da seguinte maneira:

```
1  Pegue a lista telefônica
2  Abra no meio da lista
3  Observe a página
4  Se a pessoa estiver na página
5      Ligue para a pessoa
6  Senão, se a pessoa estiver antes
7      Abra no meio da metade esquerda
8      Volte para a linha 3
9  Senão, se a pessoa estiver depois
10     Abra no meio da metade direita
11     Volte para a linha 3
12 Senão
13     Pare
```

O pseudocódigo é importante por pelo menos duas razões. Primeiro, ele permite estruturar e testar a lógica da solução antes de traduzi-la para uma linguagem formal. Segundo, ele facilita a comunicação do raciocínio para outras pessoas, mesmo que elas não dominem uma linguagem de programação específica.

Observe algumas características desse exemplo:

- Algumas linhas começam com verbos como "pegue", "abra" ou "observe". Essas ações correspondem, em programação, a `funções` ou procedimentos.
- Há instruções como "se" e "senão", que representam estruturas `condicionais`, isto é, decisões baseadas em determinadas condições.
- Certas afirmações podem ser avaliadas como verdadeiras ou falsas, como "a pessoa está antes no livro". Essas são chamadas de expressões `booleanas`.
- Instruções como "volte para a linha 3" representam estruturas de repetição, conhecidas como `loops` (ou laços).

Esses elementos - funções, condicionais, expressões booleanas e laços de repetição - constituem os blocos fundamentais da programação. No contexto do Python, linguagem utilizada neste curso, aprenderemos a implementar cada um desses componentes de maneira formal e estruturada.

1.8 Conclusão

Neste capítulo, foram apresentados os fundamentos básicos sobre o funcionamento de um sistema computacional, incluindo arquitetura, memória, organização de arquivos, interação com o sistema operacional e o papel das linguagens de programação. Esses conceitos fornecem a base necessária para compreender como o computador executa instruções e por que certas limitações surgem na prática.

A partir daqui, o curso passa a utilizar Python como linguagem de trabalho. Python é uma linguagem de alto nível, interpretada e multiparadigma que é amplamente empregada em análise de dados e pesquisa empírica em Economia, pois permite expressar soluções de forma clara e concisa, além de oferecer um amplo conjunto de bibliotecas científicas. No próximo capítulo, iniciaremos o trabalho prático: escrever código em Python, executar programas, interpretar erros e manipular dados diretamente no computador.

1.9 Exercícios

1. Um arquivo de dados possui 10 milhões de linhas e 80 colunas, com todas as células contendo valores numéricos. Suponha que cada valor numérico ocupe 8 bytes na memória.

    a. Faça uma estimativa aproximada da quantidade de memória RAM (em GB) necessária para carregar esse arquivo inteiro na memória.

b. Explique por que um computador com pouca memória RAM pode ter dificuldade para trabalhar com esse arquivo, mesmo que o arquivo "caiba" no disco rígido.

2. Sobre o sistema binário responda:

   a. Converta o número decimal 13 para a representação binária.
   b. Converta o número binário 101101 para decimal.
   c. Argumente, em poucas linhas, sobre a vantagem de computadores utilizarem o sistema binário em vez do sistema decimal.

3. Considere a seguinte estrutura de diretórios:

```
D:/home/usuario/projeto/
  dados/
     alunos.csv
     professores.csv
     materias.csv
  scripts/
     analise.py
     amostra_final.py
  resultados/
     figura.png
     tabela.csv
```

   a. Escreva o caminho absoluto para o acessar o arquivo `materias.csv`.
   b. Suponha que você esteja no diretório scripts. Escreva o caminho relativo para acessar `materias.csv`. Dica: a pasta dados e scripts estão no mesmo nível hierárquico.

4. Considere a matemática como uma linguagem formal, com regras bem definidas de escrita (sintaxe) e de significado (semântica). Suponha que o objetivo seja calcular as raízes de uma equação do segundo grau da forma $ax^2 + bx + c = 0$. Para tal você utilizará a fórmula de Bháskara.

   a. Suponha que você opte por programar essa fórmula tal qual apresentada abaixo em uma determinada linguagem de programação. Qual seria a saída esperada? Discuta o resultado.

$$x = \frac{-b \geq \sqrt{b^2 - 4ac}}{2a}$$

   b. Suponha que você opte por programar essa fórmula tal qual apresentada abaixo em uma determinada linguagem de programação. Qual seria a saída esperada? Discuta o resultado.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2}$$

2 Primeiros passos no Python

2.1 Introdução

Criado em 1989 por Guido van Rossum, no Centro de Matemática e Ciência da Computação (CWI), na Holanda, o Python é hoje uma das linguagens de programação mais utilizadas no mundo, com forte presença em ciência de dados, estatística e pesquisa científica. Ao longo desta disciplina, ele será a ferramenta central para manipulação de dados, realização de cálculos numéricos, produção de gráficos e implementação de rotinas frequentemente empregadas em trabalhos empíricos em Economia.

Neste momento inicial do curso, o objetivo não é dominar toda a linguagem nem escrever programas complexos. O foco é operacional: aprender a executar comandos simples, compreender o funcionamento do ambiente de programação e desenvolver familiaridade com a lógica básica de interação com o computador por meio de código. É importante enfatizar que programar não significa memorizar comandos, mas aprender a ler, testar e depurar código. Erros farão parte do processo desde o início — e isso é esperado. Eles constituem uma fonte valiosa de informação sobre como o computador está interpretando as instruções fornecidas.

Ao final deste capítulo, o aluno deverá ser capaz de:

- Compreender o que é o Python e por que ele é relevante para Economia;
- Instalar o Python e suas bibliotecas;
- Entender o que são ambientes virtuais e por que são utilizados;
- Compreender o papel das IDEs;
- Executar seus primeiros comandos.

Essas competências serão utilizadas continuamente ao longo do curso e são essenciais para o avanço em tópicos posteriores, como criação de funções, manipulação de objetos, análise de dados e visualização. Antes de explorar novos comandos, contudo, é fundamental compreender o ambiente no qual o programa é executado e como ocorre a interação entre o usuário e o interpretador.

2.1.1 Pré-requisitos

Este material foi elaborado para que o aluno que acompanhou e assimilou o capítulo anterior — sobre fundamentos de computação — esteja plenamente apto a prosseguir. Não há pré-requisitos adicionais. Ainda assim, é importante estar confortável com conceitos como arquivos, diretórios, sistemas operacionais e noções básicas sobre como programas são executados em um computador.

2.2 O Que é Python?

Python é uma linguagem de programação de alto nível, interpretada e de propósito geral, amplamente reconhecida por sua simplicidade, legibilidade e versatilidade. Criada no final de 1989 no Centrum Wiskunde & Informatica (CWI), foi oficialmente lançada em 1991. Seu nome foi inspirado no grupo britânico de comédia *Monty Python*.

Ser uma linguagem de alto nível significa que Python abstrai muitos detalhes técnicos do funcionamento interno do computador, como o gerenciamento manual de memória. O fato de ser interpretada implica que o código é executado dinamicamente por um interpretador, sem necessidade de um processo prévio de compilação completa. Essas características tornam Python especialmente adequado para ensino e desenvolvimento científico.

Atualmente, o Python possui forte presença em ciência de dados, aprendizado de máquina e inteligência artificial, desenvolvimento web, automação e pesquisa acadêmica.

Como qualquer ferramenta, o Python possui qualidades e limitações. Entre seus principais pontos fortes estão:

- **Simplicidade sintática e legibilidade**: o código tende a ser claro e próximo de um pseudocódigo executável. A indentação obrigatória para definir blocos reforça boas práticas de organização.
- **Alta produtividade**: é possível escrever programas funcionais com poucas linhas de código.
- **Ecossistema robusto de bibliotecas**: há ferramentas consolidadas para computação numérica (NumPy), manipulação de dados (Pandas), visualização (Matplotlib), métodos científicos (SciPy) e aprendizado de máquina (scikit-learn), entre muitas outras.
- **Portabilidade**: funciona em Windows, macOS e Linux com pouca ou nenhuma modificação.
- **Comunidade ativa**: possui uma das comunidades mais ativas do mundo da programação, com vasta produção de conteúdo, bibliotecas de código aberto e fóruns especializados.

Além do ambiente acadêmico, o Python é amplamente utilizado por grandes empresas de tecnologia e finanças, como Google, Netflix, Uber e Goldman Sachs.[1] Diversos indicadores

[1]Fonte: https://brainstation.io/career-guides/who-uses-python-today.

internacionais apontam o crescimento consistente da linguagem ao longo dos últimos anos. Índices como o *TIOBE Programming Community Index* frequentemente posicionam o Python entre as linguagens mais utilizadas no mundo.

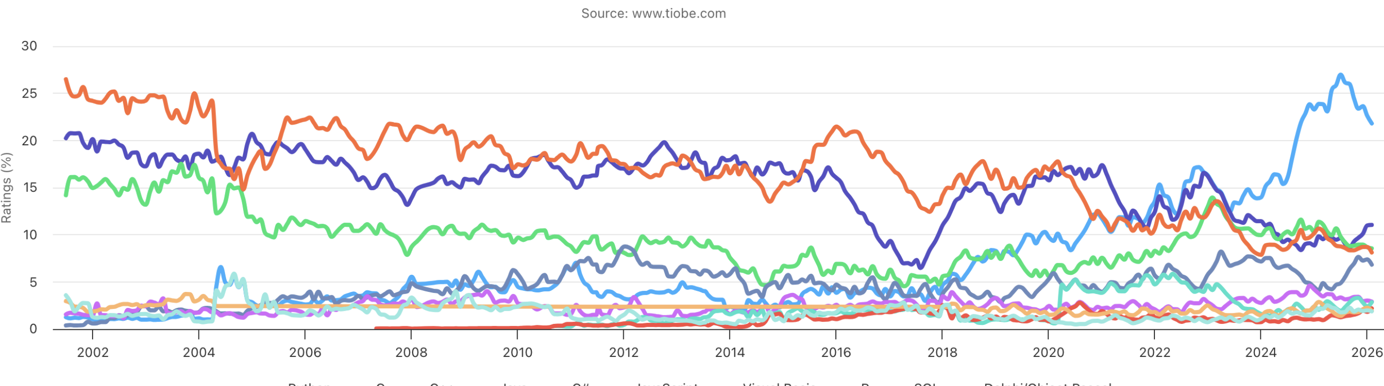


Figura 2.1: TIOBE Programming Community Index. Fevereiro de 2026

A principal limitação do Python está relacionada ao desempenho. Por ser interpretado e operar em alto nível de abstração, tende a ser mais lento que linguagens compiladas como C ou C++. No entanto, essa diferença raramente é decisiva em aplicações típicas de Economia e ciência de dados. Muitas bibliotecas numéricas utilizam implementações internas otimizadas em C, C++ ou Fortran, enquanto Python atua como interface de alto nível — combinando facilidade de uso com desempenho satisfatório. Aplicações que exigem controle extremamente fino de memória ou execução em tempo real podem se beneficiar de linguagens de nível mais baixo. Em Economia aplicada, contudo, esses casos são raros.

2.2.1 Por que Python em um Curso de Economia?

Neste curso, Python será utilizado como ferramenta central para:

- Manipulação e limpeza de bases de dados;
- Construção e transformação de variáveis;
- Implementação de procedimentos estatísticos;
- Simulações numéricas;
- Visualização de dados;
- Organização de rotinas empíricas de forma replicável.

O objetivo não é formar programadores profissionais, mas fornecer ao aluno uma ferramenta robusta que amplie sua capacidade analítica. Aprender Python, nesse contexto, significa aprender a estruturar raciocínios quantitativos de forma clara, reproduzível e escalável — três atributos fundamentais na pesquisa aplicada.

Nos próximos tópicos, passaremos da discussão conceitual para a prática: instalação, ambiente de execução e primeiros comandos.

2.3 Como Instalar o Python?

Existem diferentes formas de instalar o Python e diversas maneiras de interagir com a linguagem. Apresentamos primeiro a instalação pelo site oficial e, em seguida, a alternativa recomendada para fins acadêmicos e ciência de dados: o Anaconda.

2.3.1 Instalação pelo Site Oficial

A forma mais simples é fazer o download diretamente no site oficial. A instalação oficial fornece o interpretador do Python e um conjunto básico de bibliotecas padrão. No entanto, bibliotecas amplamente utilizadas em áreas como ciência de dados — por exemplo, NumPy e Pandas — não vêm instaladas por padrão e precisam ser adicionadas posteriormente via gerenciador de pacotes. Além disso, dependendo do sistema operacional, pode ser necessário realizar configurações adicionais para que o Python seja reconhecido no terminal.

Passo a passo:

1. Acesse o site oficial.
2. Clique na opção de Downloads.
3. Baixe o instalador correspondente ao seu sistema operacional (Windows, macOS ou Linux). O site geralmente identifica automaticamente o seu sistema e sugere a versão mais adequada.

4. No Windows, marque a opção `Add python.exe to PATH` antes de clicar em *Install Now*. Essa é etapa é fundamental, pois permite que o Python seja executado diretamente no terminal a partir de qualquer diretório do sistema.

5. Após a instalação, abra o Prompt de Comando (ou PowerShell) e execute o seguinte comando:

```
python --version
```

Se a instalação tiver sido realizada corretamente, o terminal exibirá a versão instalada do Python. Em alguns casos, pode ser necessário reiniciar o computador para que as configurações do PATH entrem em vigor.

Embora a instalação pelo site oficial funcione perfeitamente, para fins didáticos — especialmente em cursos que envolvem ciência de dados, estatística e computação científica — recomenda-se fortemente a utilização da distribuição Anaconda. O Anaconda já inclui o Python, diversas

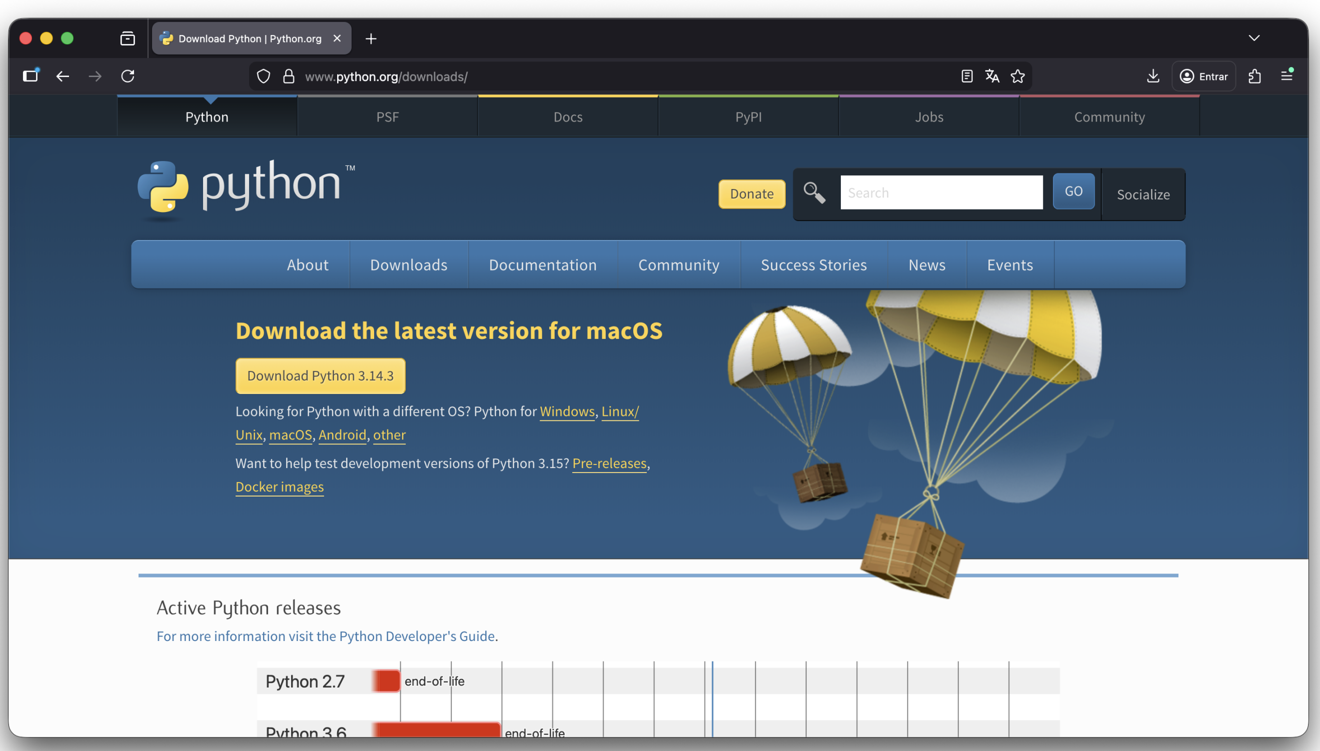


Figura 2.2: Instalação Python

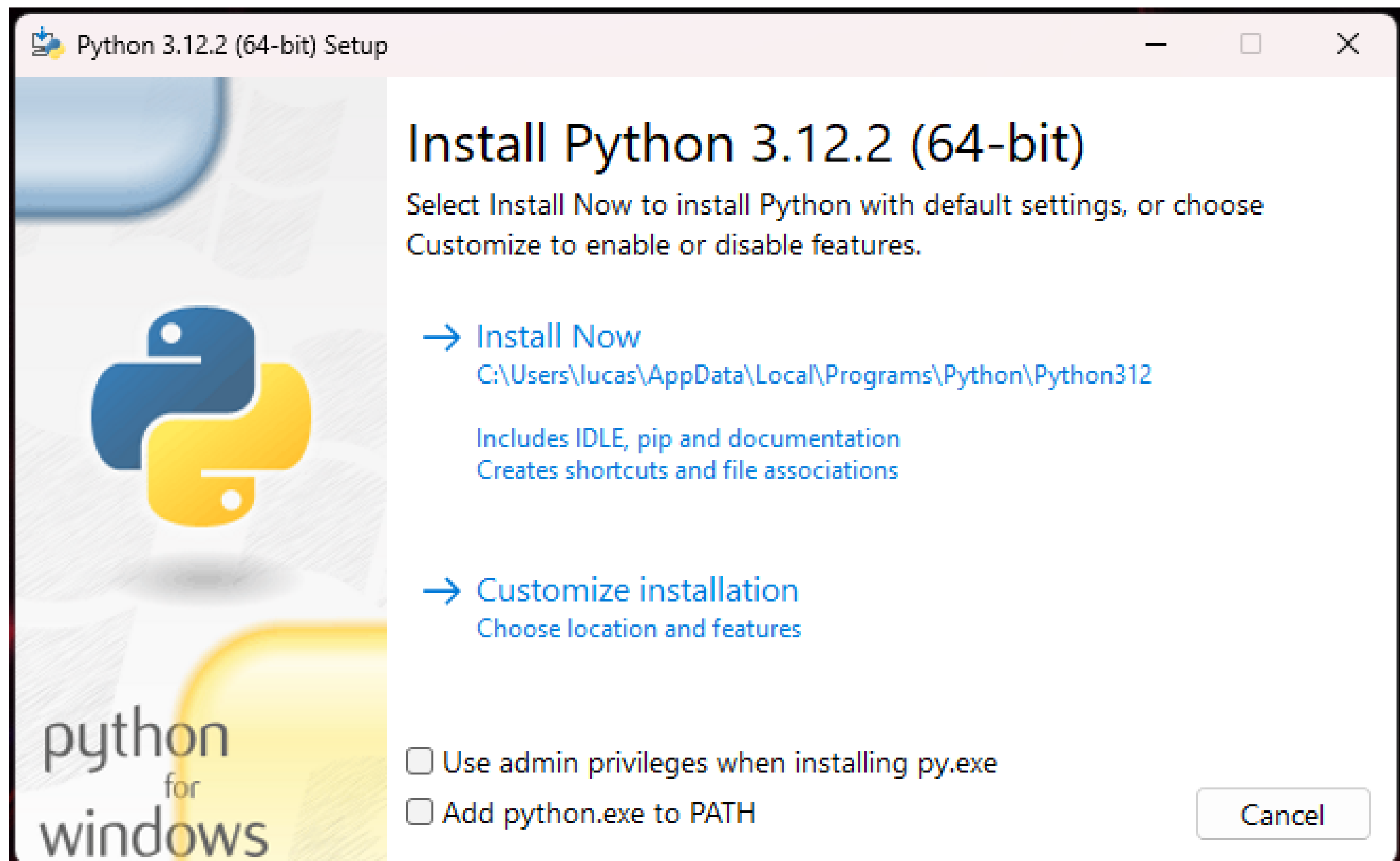


Figura 2.3: Instalação Python no Windows

bibliotecas amplamente utilizadas (NumPy, Pandas, Matplotlib, SciPy) e ferramentas úteis como o Jupyter Notebook e o Spyder. Isso reduz significativamente problemas de instalação, conflitos de versões e dificuldades iniciais, facilitando bastante o ambiente de desenvolvimento.

Nos próximos tópicos, veremos como instalar o Python via Anaconda e por que essa alternativa é, em geral, a mais conveniente para este curso.

2.3.2 Instalando via Anaconda (Recomendado)

O Anaconda é uma distribuição gratuita do Python voltada para computação científica e análise de dados. Ele simplifica significativamente a configuração do ambiente de desenvolvimento, pois já inclui, de forma integrada:

- O interpretador Python;
- Centenas de bibliotecas científicas pré-instaladas;
- O gerenciador de pacotes e ambientes `conda`;
- Ferramentas como Jupyter Notebook, JupyterLab e Spyder.

Além de tornar a instalação mais simples, o Anaconda facilita o gerenciamento de dependências e reduz conflitos entre versões de bibliotecas — algo especialmente importante em projetos acadêmicos e empíricos.

> **Nota**
>
> *O passo a passo abaixo foi elaborado com base no Windows 10. Em macOS e Linux, o procedimento é bastante semelhante, podendo haver pequenas diferenças nas telas do instalador.*

1. Acesse o site oficial do Anaconda. Escolha a versão compatível com o seu sistema operacional (Windows, macOS ou Linux) e faça o download do instalador correspondente.

1. Após concluir o download, clique duas vezes no arquivo baixado e siga as instruções do assistente de instalação. Em geral, recomenda-se manter as opções padrão sugeridas pelo instalador. Caso a instalação seja realizada para todos os usuários do computador, pode ser necessária permissão de administrador.

Após a instalação, o Python e as principais bibliotecas científicas estarão prontos para uso. O `Anaconda Navigator` pode ser acessado pelo menu Iniciar e oferece uma interface gráfica para abrir aplicativos (como Jupyter e Spyder), instalar pacotes e gerenciar ambientes de forma intuitiva.

Já o `Anaconda Prompt` permite executar comandos diretamente no terminal, sendo especialmente útil para criar e gerenciar ambientes virtuais, instalar bibliotecas e utilizar o gerenciador de pacotes `conda`.

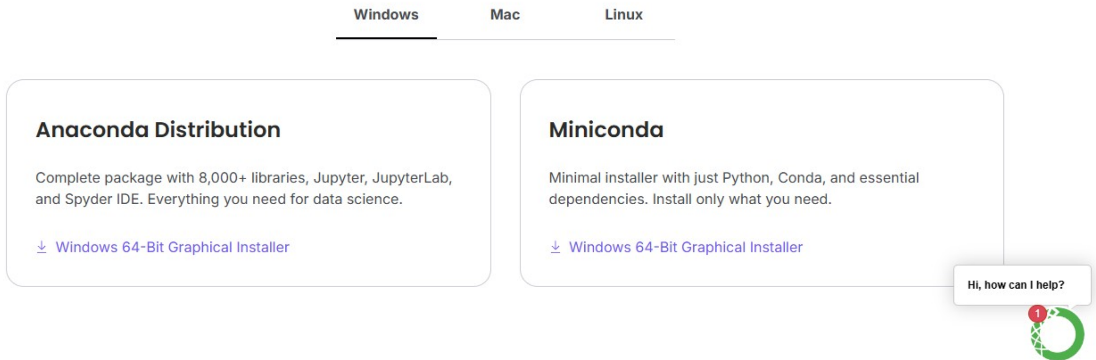


Figura 2.4: Tela de download da distribuição Anaconda. Janeiro de 2026

2.3.3 Conhecendo o Anaconda

Como mencionado anteriormente, o Anaconda é uma distribuição do Python voltada para computação científica — incluindo áreas como ciência de dados e análise estatística. Seu principal objetivo é simplificar o gerenciamento e a instalação de pacotes, tornando o ambiente mais estável e fácil de configurar.

O controle de versões e a instalação de bibliotecas no Anaconda são realizados por meio do `conda`, seu gerenciador de pacotes e ambientes. Com o conda, é possível:

- Criar ambientes virtuais isolados;
- Instalar versões específicas de bibliotecas;
- Evitar conflitos entre dependências;
- Reproduzir ambientes utilizados em diferentes projetos.

Essa funcionalidade é especialmente importante em contextos acadêmicos e profissionais, nos quais a reprodutibilidade dos resultados é fundamental.

Além do gerenciamento via terminal, o Anaconda inclui o `Anaconda Navigator`, uma interface gráfica intuitiva (*user-friendly*) que permite administrar pacotes e aplicativos sem a necessidade de utilizar comandos no terminal.

Ao abrir o Anaconda Navigator no Windows, a tela inicial será semelhante à apresentada abaixo:

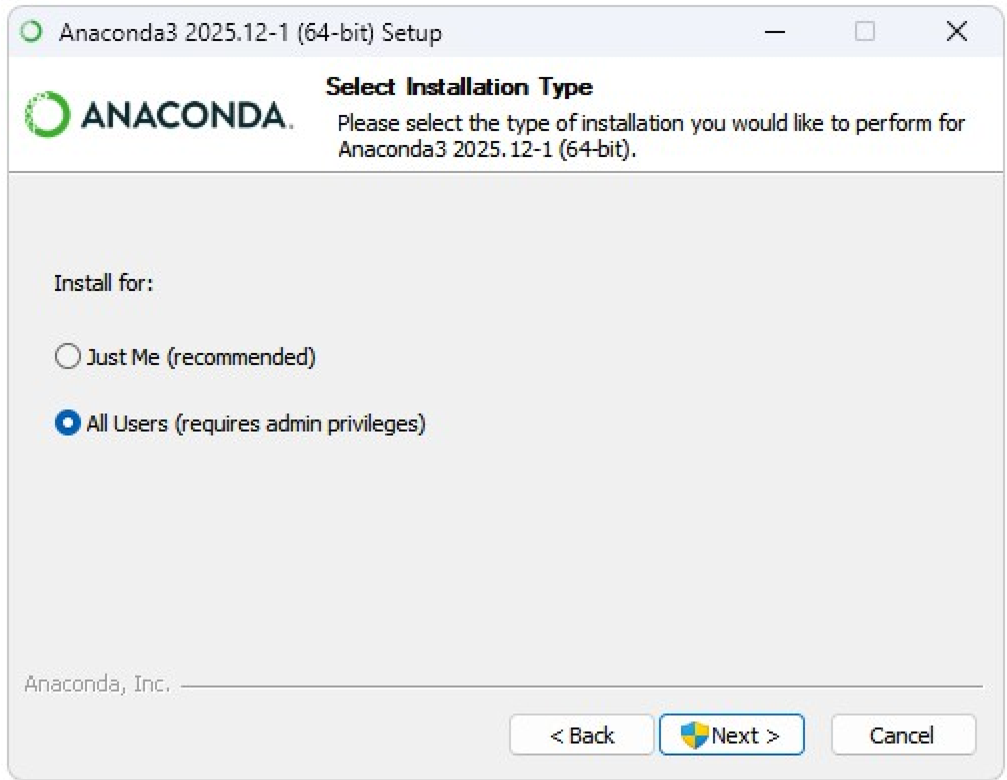


Figura 2.5: Tela de instalação da distribuição Anaconda. Janeiro de 2026

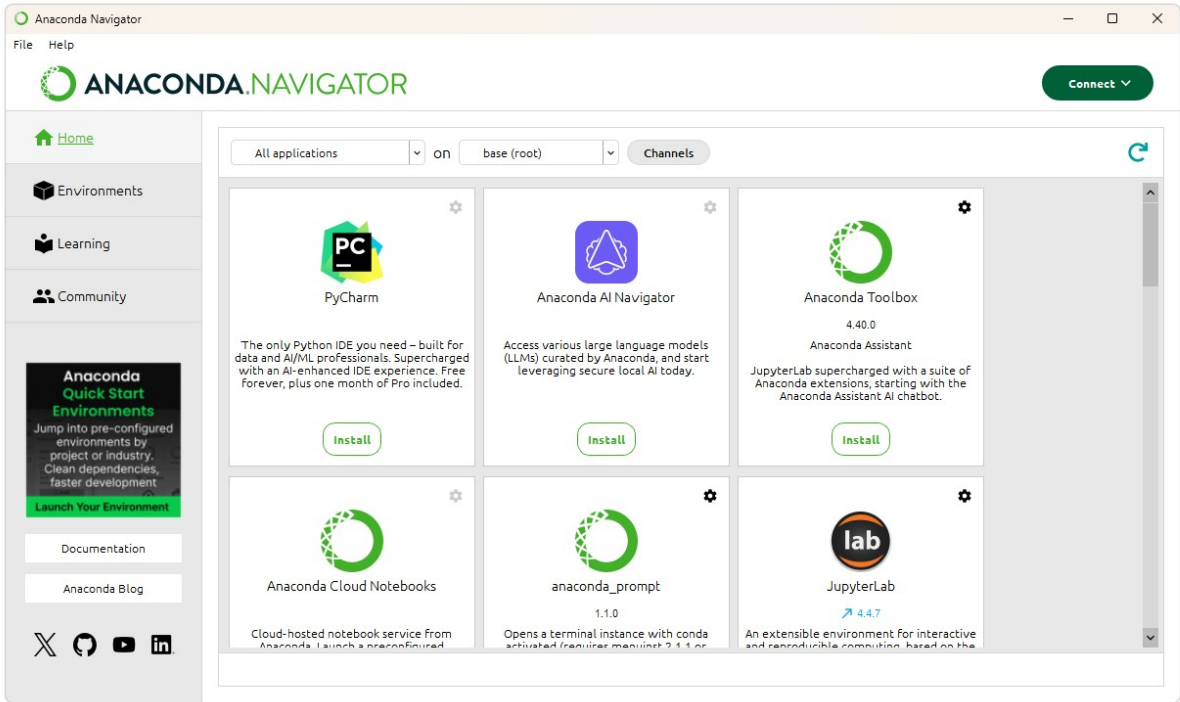


Figura 2.6: Tela inicial do Anaconda Navigator. Fevereiro de 2026

Por meio do Navigator, é possível:

- Criar e gerenciar ambientes;
- Instalar e atualizar bibliotecas;
- Abrir ferramentas como Jupyter Notebook, JupyterLab e Spyder;
- Visualizar os pacotes instalados em cada ambiente.

Embora, ao longo do curso, também utilizemos comandos no terminal, o Navigator é uma alternativa prática para quem prefere uma interface gráfica. Para mais informações sobre funcionalidades avançadas do Anaconda, a documentação oficial é um bom ponto de partida.

2.3.4 Gerenciamento de ambientes

Ambientes (*environments*) em Python são diretórios isolados que contêm um interpretador específico do Python, a biblioteca padrão correspondente e um conjunto próprio de pacotes instalados para um determinado projeto. Eles são fundamentais para gerenciar dependências específicas de cada projeto, evitar conflitos entre versões de bibliotecas, impedir interferência com a instalação global do Python e garantir portabilidade e reprodutibilidade. Criar ambientes separados não é apenas uma boa prática — é praticamente uma necessidade em projetos acadêmicos e profissionais.

Para visualizar seus ambientes no Anaconda Navigator, selecione `Environments` no menu lateral. Cada ambiente listado representa um espaço isolado com suas próprias dependências.

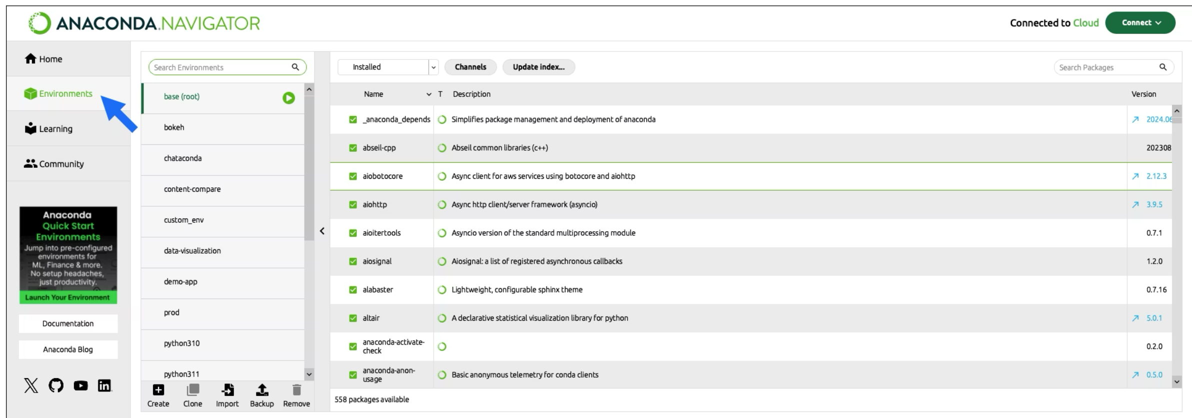


Figura 2.7: Gerenciamento de ambientes no Anaconda Navigator. Fevereiro de 2026

Use a caixa `Search Environments` para localizar um ambiente pelo nome. Você pode digitar o nome completo ou apenas parte dele para filtrar a lista e exibir apenas os ambientes que contenham a sequência de caracteres digitada.

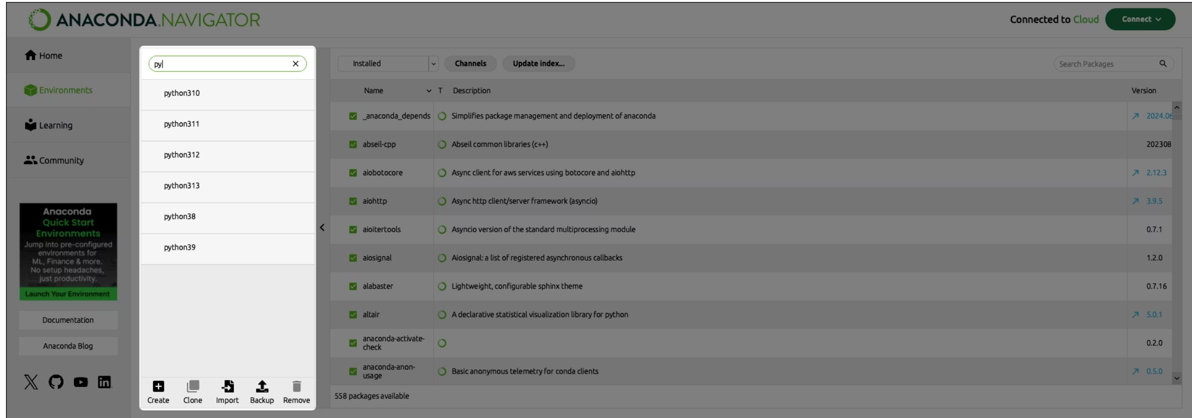


Figura 2.8: Buscando ambientes no Anaconda Navigator. Fevereiro de 2026

Na parte inferior da lista de ambientes, o Anaconda Navigator disponibiliza botões para criar, clonar, importar, exportar, fazer backup e remover ambientes.

Para criar um novo ambiente:

1. Clique em **Create** (ícone "+") na parte inferior da lista de ambientes.
2. No diálogo *Create new environment*, informe um nome descritivo para o novo ambiente.
3. Selecione Python como interpretador.

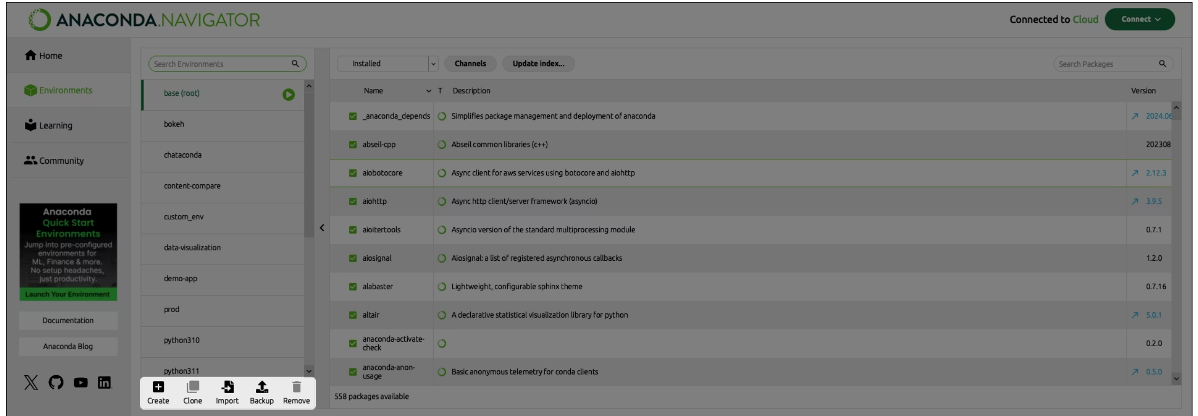


Figura 2.9: Gerenciando ambientes no Anaconda Navigator. Fevereiro de 2026

4. Escolha a versão desejada do Python no menu suspenso.
5. Clique em **Create**.

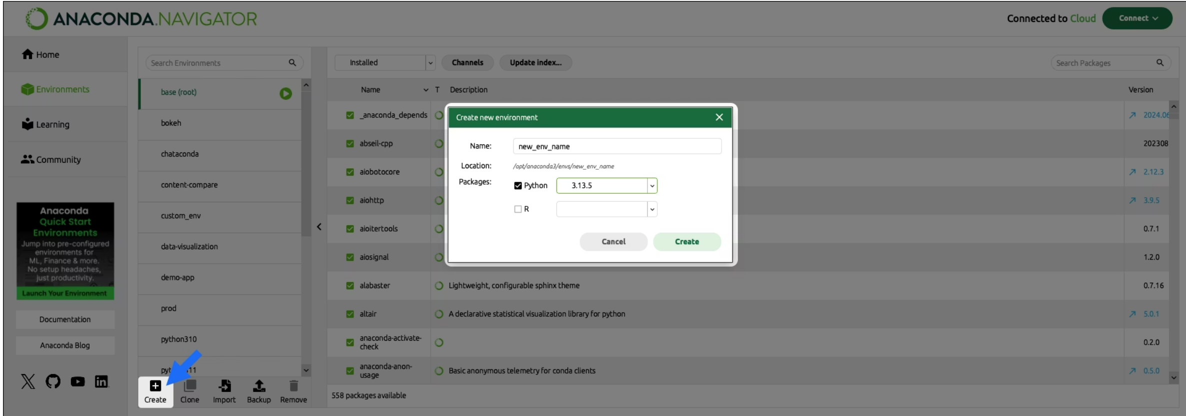


Figura 2.10: Criando ambientes no Anaconda Navigator. Fevereiro de 2026

! Importante

Por que escolher a versão do Python? Pacotes gerenciados pelo conda são construídos para versões específicas do interpretador. Escolher a versão adequada garante compatibilidade entre as bibliotecas instaladas e o ambiente de execução.

2.3.5 Instalando Pacotes

Você pode expandir as funcionalidades do seu ambiente Python instalando pacotes de duas formas principais:[2]

- **Anaconda Navigator**: interface gráfica (clique e seleção), recomendada para quem prefere evitar o uso do terminal.
- **Conda via linha de comando (Anaconda Prompt no Windows ou Terminal no macOS/Linux)**: método mais rápido, flexível e adequado para automação e projetos mais avançados.

Para instalar novos pacotes utilizando o terminal, abra o **Anaconda Prompt**, que já vem configurado para reconhecer o comando `conda`. Como exemplo, vamos instalar o pacote `tqdm`, utilizado para criar barras de progresso em loops e tarefas repetitivas.

1. Abra o terminal adequado: No Windows, use o Anaconda Prompt; no macOS ou Linux, utilize o terminal padrão.
2. Ative seu ambiente (Recomendado): Evite instalar pacotes no ambiente base para manter o sistema organizado.

```
conda activate nome_do_ambiente
```

3. Antes de instalar, verifique a existência do pacote:

```
conda list tqdm
```

4. Se o pacote não for listado, utilize o gerenciador `conda` para instalar o pacote:

```
conda install tqdm
```

Após a instalação, o pacote estará disponível para uso no ambiente ativo.

> **Nota**
>
> Dica: Caso o pacote não seja encontrado nos canais padrão do Conda, você pode recorrer ao `pip install tqdm`.

Uma das grandes vantagens do Anaconda é o seu ecossistema "out-of-the-box": as principais bibliotecas científicas — como NumPy, SciPy, Matplotlib e Pandas — já vêm pré-instaladas. Diferente da instalação "pura" do Python, onde cada pacote deve ser adicionado manualmente, o Anaconda oferece um ambiente pronto para análise de dados desde o primeiro uso. Além disso, o comando `conda list` funciona como um inventário completo, exibindo não apenas os

[2] A documentação oficial da Anaconda sobre instalação de pacotes pode ser consultada neste link.

pacotes instalados, mas também suas versões específicas, o canal de origem e o ambiente ativo no momento.

Entendendo a Diferença: `conda` vs `pip`

Embora ambos instalem bibliotecas, eles operam com filosofias diferentes. Entender quando usar cada um evita conflitos que podem "quebrar" seu ambiente de trabalho.[3]

- `pip` (*Python Package Index*): É o gerenciador padrão e específico da linguagem Python. Ele foca em instalar bibliotecas Python disponíveis no repositório PyPI. É a escolha ideal quando você precisa de uma biblioteca muito específica ou muito recente que ainda não foi homologada pelo Anaconda.
- `conda` (*Anaconda/Conda-forge*): É um gerenciador de ambientes e pacotes agnóstico à linguagem. Além de Python, ele gerencia bibliotecas em R, C++, Java e dependências de sistema (como drivers da NVIDIA). O conda é mais rigoroso: ele verifica se a nova instalação causará conflitos com o que já existe no seu ambiente, garantindo maior estabilidade.

2.4 IDEs

A interação com Python ocorre por meio de um ambiente de desenvolvimento. Esses ambientes são chamados de **IDEs** (*Integrated Development Environments*), ou Ambientes de Desenvolvimento Integrado.

Uma IDE é um software que reúne, em um único local, as principais ferramentas necessárias para o desenvolvimento de programas. Seu uso torna o processo de escrita, execução e depuração de código mais eficiente, organizado e produtivo.

Existem IDEs específicas para determinadas linguagens ou finalidades, e outras mais gerais e flexíveis. Entre as mais conhecidas estão **Visual Studio Code**, **Sublime Text** e **PyCharm**.

2.4.1 Componentes típicos de uma IDE

Embora variem em funcionalidades e interface, as IDEs geralmente incluem os seguintes elementos:

- **Editor de texto**: Permite escrever e editar programas com recursos como destaque de sintaxe, identação automática e autocompletar, facilitando a organização e a legibilidade do código.

[3] Para conhecer mais sobre a diferença entre esses dois métodos de instalação de pacotes, confira estes artigos sobre a arquitetura do Conda e estratégias de instalação: https://www.anaconda.com/blog/understanding-conda-and-pip e https://pythonspeed.com/articles/conda-vs-pip/

- **Ferramentas integradas de execução/compilação**: Responsáveis por compilar (em linguagens compiladas) ou executar o código diretamente no ambiente, permitindo testar o programa sem sair da IDE.
- **Depurador integrado**: Ferramenta que auxilia na identificação e correção de erros, possibilitando executar o código passo a passo e inspecionar variáveis durante a execução.

> **ℹ Nota**
>
> Em linguagens compiladas, a IDE também integra um compilador. No caso do Python, que é interpretado, o foco está na integração com o interpretador.

O uso de uma IDE traz diversas vantagens:

- Reduz o tempo de desenvolvimento;
- Facilita a organização do código;
- Aumenta a produtividade;
- Ajuda a identificar erros com maior rapidez;
- Centraliza ferramentas essenciais em um único ambiente.

2.4.2 Spyder

O Spyder é uma IDE de código aberto voltada especialmente para computação científica e análise de dados. Trata-se de uma ferramenta leve, simples e, ao mesmo tempo, bastante poderosa.

Entre seus principais recursos, destacam-se:

- Editor com destaque de sintaxe e organização automática do código;
- Console interativo integrado;
- Depurador (debugger) eficiente;
- Explorador de variáveis, que permite visualizar objetos armazenados na memória.

O explorador de variáveis é particularmente útil para iniciantes, pois evita a necessidade de imprimir repetidamente variáveis no console para inspecionar seus conteúdos. O Spyder é instalado automaticamente com o Anaconda, o que o torna uma opção prática e acessível para uso acadêmico.

2.4.3 Jupyter Notebook

O Jupyter Notebook é amplamente utilizado em ciência de dados, pesquisa e ensino. Diferentemente de uma IDE tradicional baseada em arquivos .py, o Jupyter organiza o trabalho em notebooks, compostos por células. Essas células podem conter:

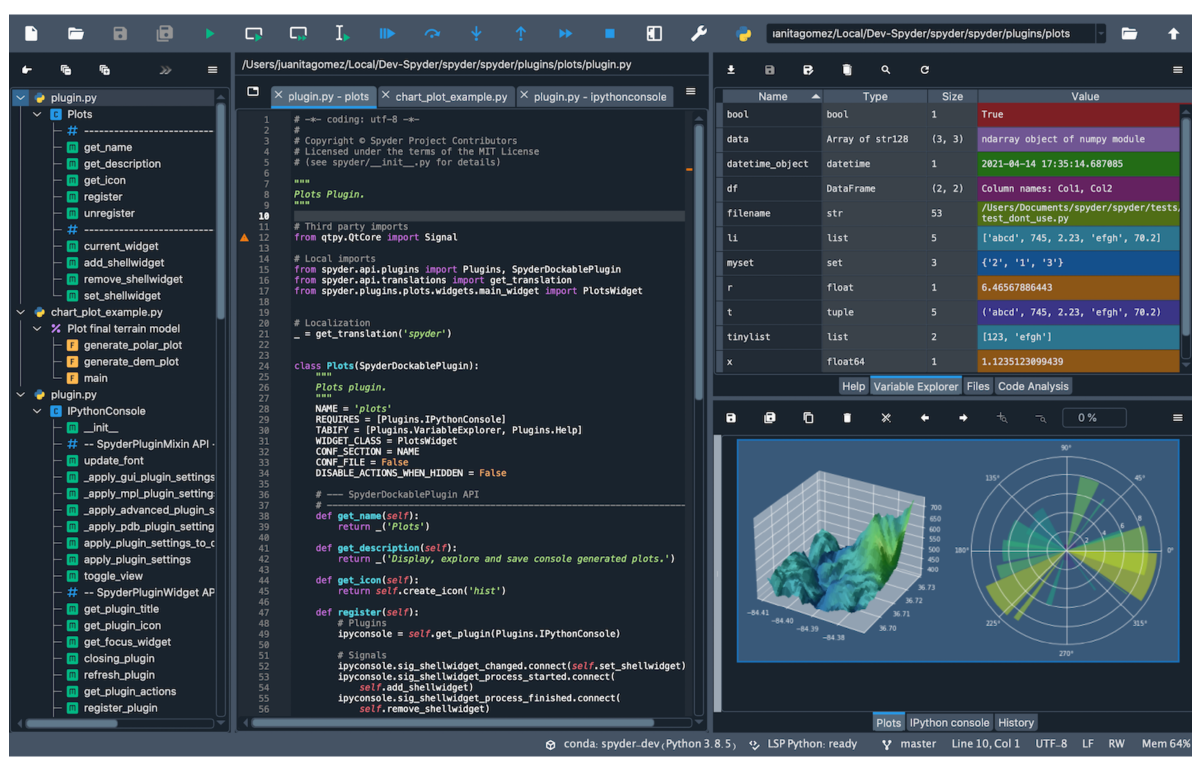


Figura 2.11: Spyder. Fevereiro de 2026

- Código executável;
- Texto explicativo em Markdown;
- Fórmulas matemáticas (LaTeX);
- Gráficos e visualizações.

Essa estrutura permite intercalar explicação e execução, tornando o código mais claro, documentado e replicável. Por isso, o Jupyter é particularmente adequado para análise exploratória de dados, relatórios técnicos, projetos acadêmicos e material didático. Grande parte do material deste curso foi desenvolvida em Jupyter, justamente por sua capacidade de integrar texto, código e resultados em um único documento!

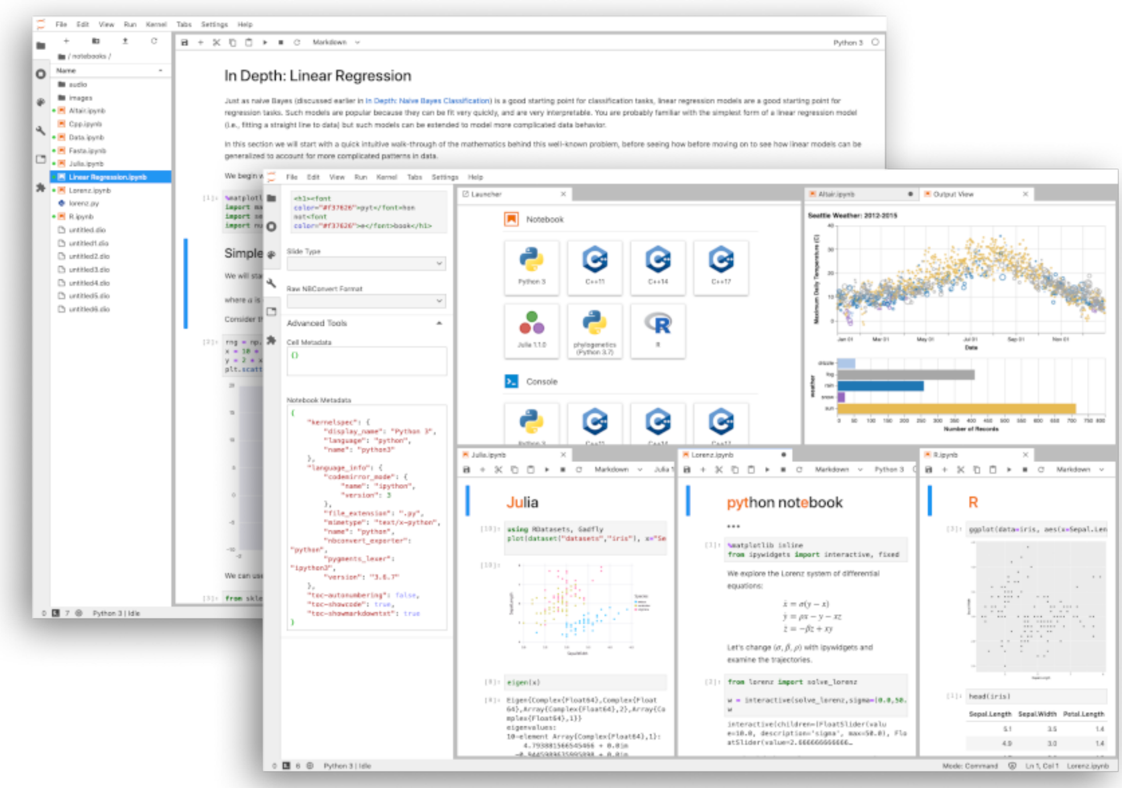


Figura 2.12: Jupyter Notebook. Fevereiro de 2026

2.4.4 Google Colaboratory (Colab)

O Google Colaboratory (Colab) é um ambiente baseado em nuvem, mantido pelo Google, que permite executar notebooks diretamente no navegador, sem necessidade de instalação local. Assim como o Jupyter, o Colab permite combinar:

- Código Python;
- Texto em Markdown;
- Imagens e gráficos;
- Resultados das execuções.

A principal diferença é que, no Colab, o código é executado em servidores do Google — e não no seu computador. Isso pode ser vantajoso para tarefas mais pesadas, especialmente quando se utiliza GPU. A versão gratuita possui limitações de memória e tempo de execução, enquanto planos pagos oferecem maior capacidade computacional.

O Colab é uma excelente alternativa para quem não deseja instalar o Python localmente, utiliza múltiplos computadores ou precisa compartilhar notebooks com facilidade.

Figura 2.13: Google Colab. Fevereiro de 2026

2.4.5 VS Code

O Visual Studio Code (VS Code) é um editor de código altamente flexível e extensível, amplamente utilizado tanto por iniciantes quanto por desenvolvedores profissionais. Embora

não seja exclusivamente voltado para Python, ele pode ser configurado como uma IDE completa por meio da instalação da extensão oficial de Python.

As principais vantagens são:

- Interface moderna e altamente personalizável;
- Integração com Git e controle de versões;
- Suporte a múltiplas linguagens;
- Excelente sistema de extensões;
- Integração com Jupyter Notebooks.

O VS Code é particularmente interessante para alunos que desejam utilizar Python em projetos mais amplos ou que pretendem avançar para desenvolvimento de software ou aplicações mais complexas.

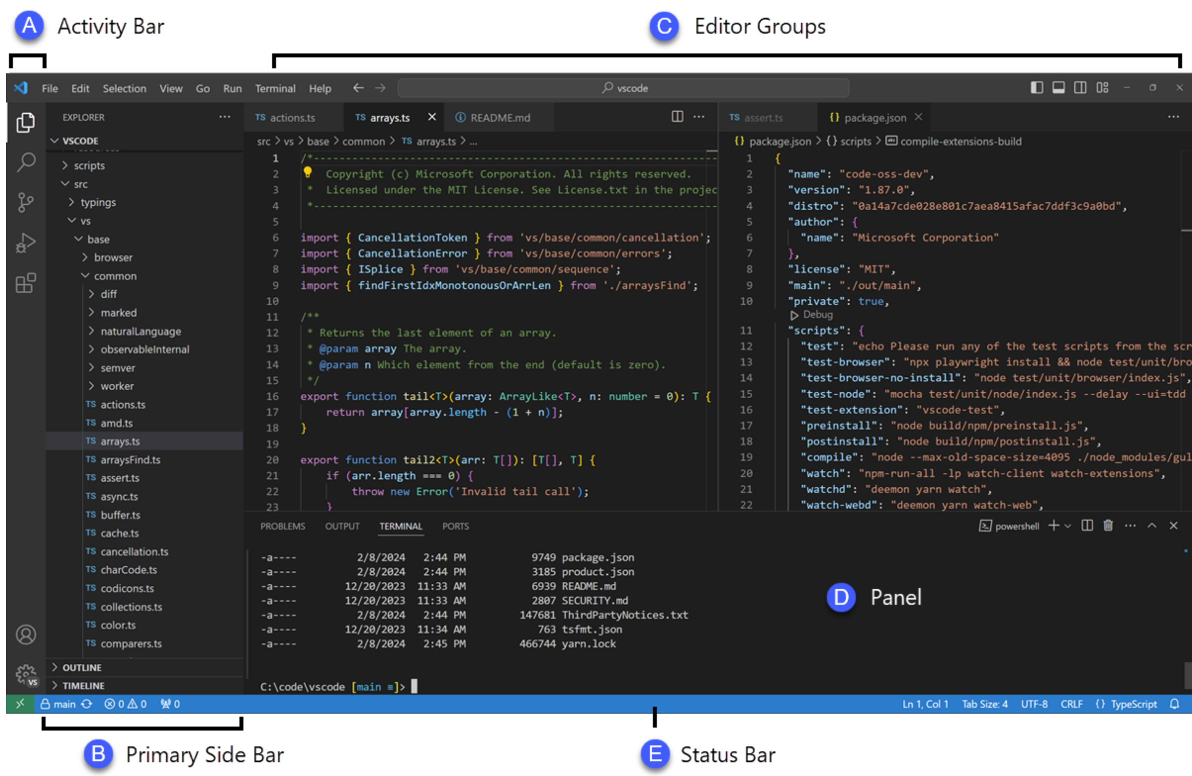


Figura 2.14: VS Code. Fevereiro de 2026

2.4.6 Positron

O Positron é uma ferramenta mais recente desenvolvida pela Posit (antiga RStudio), projetada para oferecer um ambiente moderno voltado à ciência de dados. Ele combina características

de:

- IDE tradicional;
- Ambiente interativo;
- Integração com notebooks;
- Ferramentas avançadas para análise de dados.

Seu foco está na produtividade em projetos de dados, com forte integração a fluxos de trabalho científicos. Embora ainda esteja em expansão, representa uma alternativa promissora para quem trabalha com Python (e também com R) em contextos analíticos.

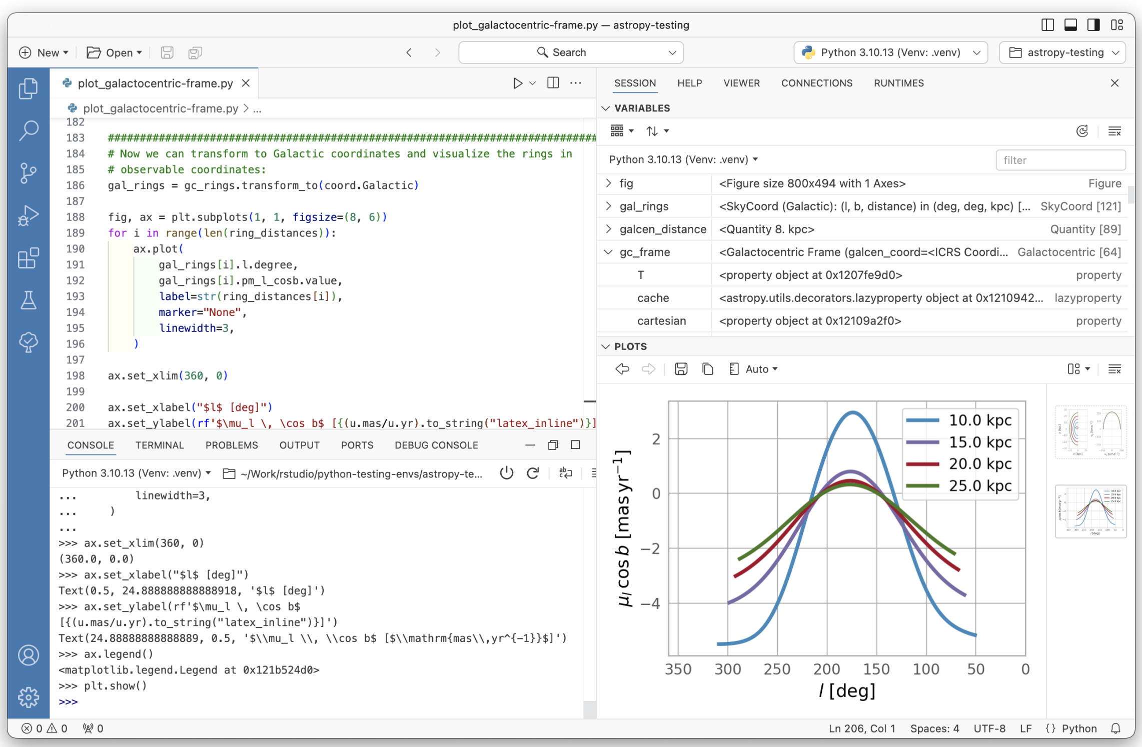


Figura 2.15: Positron. Fevereiro de 2026

2.4.7 Qual Escolher?

Para este curso, recomenda-se prioritariamente:

- Jupyter Notebook (principal ferramenta didática);
- Spyder (para quem prefere uma IDE tradicional);
- Colab (para quem não deseja instalar nada localmente).

Ferramentas como VS Code e Positron são excelentes opções para projetos mais avançados. Independentemente da escolha, o mais importante não é a ferramenta em si, mas a capacidade de organizar o raciocínio computacional de forma clara, estruturada e replicável.

2.5 Markdown

Markdown é uma linguagem de marcação leve (*lightweight markup language*) utilizada para adicionar formatação a documentos de texto simples (plain text). Criada por John Gruber em 2004, tornou-se uma das linguagens de marcação mais populares do mundo. Diferentemente de editores do tipo `WYSIWYG` (como o Microsoft Word), nos quais você clica em botões para aplicar formatação visual imediata, no Markdown a formatação é feita por meio de símbolos inseridos diretamente no texto. Esses símbolos indicam como o conteúdo deve ser exibido quando o documento for processado. A marcação em si não aparece no resultado final — apenas o texto formatado.

Pode parecer mais simples utilizar um editor tradicional para formatar textos, mas o Markdown oferece diversas vantagens:

1. Arquivos em Markdown são apenas texto simples. Isso significa que podem ser abertos em qualquer editor de texto, não dependem de formatos proprietários (como `.docx`) e são fáceis de compartilhar entre sistemas operacionais.
2. Você pode criar e editar arquivos Markdown em qualquer sistema operacional (Windows, macOS, Linux) e em praticamente qualquer dispositivo.
3. Mesmo que a aplicação que você utiliza deixe de existir no futuro, seus arquivos Markdown continuarão legíveis como texto simples. Isso é especialmente importante para documentos acadêmicos, relatórios e materiais que precisam ser preservados por longos períodos.
4. Markdown pode ser facilmente convertido em HTML, a linguagem utilizada pelos navegadores.

Ao longo do curso, utilizaremos o Jupyter Notebook para trabalhar com Python e Markdown. Sem Markdown, as células do Jupyter exibiriam apenas código. Embora seja possível usar comentários no Python (`# comentário`), eles não permitem formatação avançada, não suportam links ou imagens e não estruturam o texto visualmente.

O Markdown resolve esse problema, permitindo alternar naturalmente entre explicações teóricas, código executável e resultados e gráficos. Abaixo estão os principais elementos da sintaxe básica[4]:

Elemento	Sintaxe Markdown
Título (Cabeçalho)	# Título 1## Título 2### Título 3
Negrito	`**texto em negrito**`

[4]Consulte o guia oficial para mais informações sobre como usar o Markdown.

Elemento	Sintaxe Markdown
Itálico	`*texto em itálico*`
Citação (Bloco de citação)	`> texto citado`
Lista numerada	1. Primeiro item2. Segundo item
Lista com marcadores	- Primeiro item- Segundo item
Código (em linha)	`código`
Linha horizontal	`---`
Link	`[texto](https://www.exemplo.com)`
Imagem	`![alt](imagem.jpg)`

2.6 Construindo o Primeiro Programa

Uma vez compreendido o papel do ambiente de programação, o próximo passo é escrever e executar instruções simples. Tradicionalmente, o primeiro programa em qualquer linguagem exibe a mensagem "*Hello, World!*". Em Python, ele é escrito da seguinte forma:

```
print("Hello, World!")
```

```
Hello, World!
```

Esse é um exemplo do uso da função `print()`. Apesar do nome, ela não "imprime" nada em papel; sua função é exibir um resultado na tela. Alguns pontos importantes:

- As aspas delimitam o texto que será exibido. Elas não aparecem no resultado.
- Os parênteses indicam que print é uma função.
- O conteúdo dentro dos parênteses é o argumento da função.

Exploraremos funções com mais detalhes em breve. Por ora, o importante é entender que estamos fornecendo uma instrução ao interpretador e ele a executa imediatamente.

2.6.1 Usando Python como uma Calculadora

Além de exibir texto, o interpretador Python pode ser utilizado como uma calculadora simples. Basta digitar uma expressão e o resultado será apresentado. Os operadores aritméticos básicos (adição, subtração, multiplicação e divisão) são: `+`, `-`, `*` e `/`. Por exemplo:

```
2 + 2
```

```
4
```

```
50 - 10
```

```
40
```

```
8 * 5
```

```
40
```

```
20 / 4
```

```
5.0
```

O operador `**` é utilizado para calcular potências:

```
10 ** 2
```

```
100
```

Em algumas linguagens, o símbolo `^` representa a exponenciação. No Python, entretanto, `^` é um operador *bitwise* (XOR).

```
6 ^ 2
```

```
4
```

O resultado pode parecer inesperado caso você não esteja familiarizado com operações bit a bit. Não abordaremos operadores bitwise neste curso, pois eles não são relevantes para nossos objetivos imediatos.[5]

Note que parênteses `()` podem ser usados para agrupar expressões. Por exemplo,

```
(50 - 5 * 6) / 4
```

```
5.0
```

[5] Saiba mais sobre operadores bitwise em http://wiki.python.org/moin/BitwiseOperators.

Em Python, números inteiros (por exemplo, 2, 4 e 20) são do tipo `int`. Já os números com parte fracionária (por exemplo, 5.0 e 1.6) são do tipo `float`. Veremos mais sobre tipos de dados ao longo do curso.

Por enquanto, é válido mencionar que a divisão tradicional (`/`) sempre retorna um número do tipo ponto flutuante (`float`), mesmo quando o resultado é inteiro:

```
# divisão clássica retorna um ponto flutuante
17 / 3
```

```
5.666666666666667
```

Para realizar uma divisão inteira (divisão pelo piso) e receber um inteiro como resultado, você pode utilizar o operador `//`:

```
# divisão pelo piso descarta a parte fracionária
17 // 3
```

```
5
```

Para obter o resto da divisão, utilize `%`:

```
# o operador % retorna o resto da divisão
17 % 3
```

```
2
```

2.6.2 Instruções e variáveis

Uma instrução é a unidade básica de um programa — uma linha de código que produz algum efeito, como exibir um valor na tela ou armazenar uma informação na memória. Entre os recursos mais importantes de uma linguagem de programação está a possibilidade de trabalhar com variáveis. Uma **variável** pode ser entendida como um nome que faz referência a um valor armazenado na memória do computador. Ela funciona como um rótulo que aponta para um dado — que pode ser um número, um texto, ou estruturas mais complexas. Em termos simples:

> Uma variável é um contêiner para um valor dentro do seu programa.

No Python, criamos uma variável por meio de uma instrução de atribuição, utilizando o operador `=`:

```
nome = "Arthur"
n = 17
pi = 3.141592653589793
```

Essas três linhas fazem o seguinte:

- Atribuem uma `string` à variável nome;
- Atribuem o `número inteiro` 17 à variável n;
- Atribuem uma aproximação de π à variável pi.

> ❗ Importante
>
> O símbolo `=` aqui não significa igualdade matemática. Ele é chamado de operador de atribuição. Ele funciona da seguinte maneira: "O valor à direita é atribuído (copiado) para o nome à esquerda". Assim, a leitura correta é: "A variável `n` recebe o valor 17."

Depois de definida, uma variável pode ser utilizada em expressões:

```
n + 25
```

```
42
```

```
z = n * 2 + 16
print(z)
```

```
50
```

Diferentemente da matemática, em programação não precisamos nos limitar a nomes como `x` ou `y`. Podemos usar nomes descritivos, como `idade` ou `nome_completo`. Boas práticas para nomes de variáveis são:

- Podem conter letras e números;
- Não podem começar com número;
- Podem conter sublinhado (`_`);
- Não podem conter espaços;
- Não podem ser palavras reservadas da linguagem.

Se você der um nome ilegal a uma variável, recebe um erro de sintaxe:

```python
2026curso = "Métodos Computacionais"
```

```
SyntaxError: invalid syntax (118123792.py, line 1)
  Cell In[15], line 1
    2026curso = "Métodos Computacionais"
        ^
SyntaxError: invalid syntax
```

Esse nome é inválido porque começa com número. Outro exemplo de nome ilegal é:

```python
class = "Métodos Computacionais"
```

```
SyntaxError: invalid syntax (2049420017.py, line 1)
  Cell In[16], line 1
    class = "Métodos Computacionais"
          ^
SyntaxError: invalid syntax
```

Aqui o problema é diferente: `class` é uma palavra-chave do Python. Palavras-chave são termos reservados pela linguagem para definir sua estrutura. Elas não podem ser usadas como nomes de variáveis. Algumas palavras-chave do Python 3 são:

```
and         del         from        None        True
as          elif        global      nonlocal    try
assert      else        if          not         while
break       except      import      or          with
class       False       in          pass        yield
continue    finally     is          raise
def         for         lambda      return
```

Não é necessário memorizar essa lista. A maioria das IDEs destaca palavras-chave em cores diferentes, facilitando sua identificação.

2.6.3 Valores e Tipos

Até agora vimos três tipos de valores:

- 2 é um número inteiro;
- 42.0 é um número de ponto flutuante; e
- "Hello World!" é uma string, assim chamada porque as letras que contém estão em uma sequ6encia em cadeia.

Um tipo de valor é conhecido como **tipo**. Todo valor tem um tipo - ou, em algumas ocasiões, dizemos que ele "pertence" a um tipo específico. O Python disponibiliza uma função chamada `type`, que indica o tipo de qualquer valor.

Por exemplo, um número inteiro tem um o tipo `int`:

```
type(2)
```

```
int
```

Um número decimal, ou de ponto flutuante, tem o tipo `float`:

```
type(42.0)
```

```
float
```

E uma sequência de caracteres, ou string, tem o tipo `str`:

```
type("Hello World!")
```

```
str
```

2.7 Conclusão

Neste capítulo você aprendeu a instalar o Python, executar comandos iniciais e a navegar no ambiente de programação. Mais importante que decorar instruções é desenvolver uma atitude prática perante o código: formular hipóteses, testar, interpretar erros, corrigir e validar novamente. Nos próximos capítulos vamos explorar os objetos básicos da linguagem — números, strings e listas — sempre apoiando os novos conceitos nas habilidades operacionais adquiridas aqui.

2.8 Exercícios

1. Utilize o Python para responder as seguintes perguntas:

    a. Quantos segundos há em 42 minutos e 42 segundos?
    b. Quantas milhas há em 10 quilômetros? Dica: uma milha equivale a 1,61 quilômetro.

   c. Se você correr 10 quilômetros em 42 minutos e 42 segundos, qual é o *pace* médio (tempo por quilômetro em minutos e segundos)?

2. Crie uma variável chamada `salario_mensal` com valor igual a 3500.

   a. Crie uma variável chamada `rendimento_anual` que seja construída a partir de `salario_mensal` e considere o recebimento do 13º salário no mês de dezembro. Utilize uma instrução para exibir `rendimento_anual` na tela.
   b. Calcule e exiba na tela o valor do rendimento anual após o pagamento de impostos, considerando a incidência de uma alíquota média de 16.5%.

3. Explique, com suas próprias palavras, a diferença entre escrever código em um editor de texto simples e escrever código em uma IDE ou ambiente interativo. Sua resposta deve mencionar pelo menos dois recursos que facilitam o aprendizado em IDEs.

4. Considere o código abaixo:

```
print("Resultado:"
resultado == 10 + 5
```

   a. Execute o código e identifique o tipo e posição do(s) erro(s) no código.
   b. Explique com suas palavras o que está de errado.
   c. Corrija o(s) erro(s) e execute novamente.

3 Tipos de Dados e Expressões

3.1 Introdução

Este capítulo apresenta os elementos fundamentais com que o Python trabalha: valores, tipos de dados e objetos, além de uma introdução prática a funções, variáveis e exploração interativa do ambiente. Ao final, o aluno deverá ser capaz de:

- Identificar e usar os tipos básicos (`int`, `float`, `str`, `bool`) em programas Python.
- Compreender que tudo é um objeto em Python e saber consultar métodos e atributos.
- Manipular strings (concatenação, f-strings, escape, slicing e métodos úteis).
- Converter entre tipos e lidar com entradas de usuário.
- Usar `dir()` e `help()` para descobrir funcionalidades de objetos.

3.1.1 Pré-requisitos

Conhecimentos básicos de lógica e familiaridade com o interpretador Python (execução de scripts/linhas interativas). Neste capítulo usaremos principalmente funções nativas do Python.

3.2 Valores, Tipos e Objetos

Toda linguagem de programação precisa representar informações. Essas informações são chamadas de valores. Um valor pode ser um número, um texto, uma resposta lógica (verdadeiro ou falso), entre outros.

No entanto, o computador precisa saber como interpretar cada valor. O número 5 é diferente do texto “5”. Embora visualmente pareçam iguais, internamente são representados de forma distinta e permitem operações diferentes. É aqui que entra o conceito de tipo de dado.

Um tipo de dado define:

- Como o valor é armazenado na memória;
- Quais operações podem ser realizadas com ele;
- Como o interpretador deve tratá-lo durante a execução do programa.

Em Python, todos os valores são implementados como **objetos**. Isso significa que cada valor possui:

- Um tipo (sua categoria);
- Um valor (a informação que representa);
- Um conjunto de métodos (operações específicas que podem ser aplicadas a ele).

Essa característica torna o Python uma linguagem extremamente consistente e poderosa.

3.3 Breve Introdução às Funções

Em praticamente todas as linguagens de programação, o conceito de função é central. Funções permitem organizar o código, reutilizar instruções e estruturar programas de forma clara e modular. De maneira intuitiva, uma função pode ser entendida como uma ação — um "verbo" — que executa uma tarefa específica e, frequentemente, retorna um resultado.

Toda linguagem já possui um conjunto de funções pré-definidas, chamadas de funções nativas (ou *built-in functions*). Elas representam operações básicas que o computador já sabe realizar, como exibir algo na tela, converter tipos de dados ou calcular o tamanho de um objeto.

No nosso primeiro programa, utilizamos a função `print`. Essa função exibe informações na tela. Ela não imprime um texto fixo; imprime aquilo que for fornecido a ela.

```python
print("Olá, Mundo")
```

```
Olá, Mundo
```

3.3.1 Argumentos

Um argumento é a informação que passamos para uma função para que ela saiba sobre qual valor deve operar. No exemplo anterior, `"Olá, Mundo"` é o argumento da função `print`.

Podemos pensar da seguinte forma:

- A função é a ação;
- O argumento é o objeto sobre o qual a ação será realizada.

3.3.2 Variáveis

Para que um programa seja dinâmico, precisamos armazenar valores. É aí que entram as variáveis. Uma variável é um nome que se refere a um valor armazenado na memória. Ela funciona como um rótulo que aponta para um objeto. Exemplo:

```
curso = "Métodos Computacionais"
```

Aqui, `curso` é a variável, `"Métodos Computacionais"` é o valor armazenado e `=` é o operador de atribuição. É importante destacar que, em Python, o símbolo `=` não significa igualdade matemática, mas sim atribuição.

Quando escrevemos, `curso = "Métodos Computacionais"`, estamos dizendo: "Execute a função `input`, pegue o valor retornado por ela e armazene esse valor na variável curso."

Exploraremos o conceito de funções com maior profundidade em aulas futuras, quando analisaremos sua definição formal, parâmetros, valores de retorno e sua importância na organização e modularização de programas.

3.4 Tipos Básicos

3.4.1 Inteiros (int)

O tipo `int` representa números inteiros: positivos, negativos ou zero. Do ponto de vista matemático, são os elementos do conjunto dos números inteiros.

```
x = 10
y = -3
print(x + y)
```

```
7
```

Inteiros permitem operações aritméticas tradicionais:

- `+` soma
- `-` subtração
- `*` multiplicação
- `/` divisão
- `//` divisão inteira
- `%` resto da divisão (módulo)
- `**` potência

O operador `%` é especialmente útil em programação. Por exemplo, para verificar se um número é par:

```python
numero = 8
print(numero % 2 == 0)  # True
```

```
True
```

3.4.2 Pontos Flutuantes (float)

O tipo `float` representa números reais com parte decimal. Por exemplo:

```python
x = 3.14
y = 2.5
print(x * y)
```

```
7.8500000000000005
```

Internamente, esses números são armazenados em formato de **ponto flutuante binário**, o que implica aproximações. Isso significa que certos valores aparentemente simples podem não ser representados de forma exata.

Por exemplo:

```python
print(0.1 + 0.2)
```

```
0.30000000000000004
```

Isso não é erro do Python, mas uma limitação da representação binária de números decimais. Para lidar com isso, utilizamos, por exemplo, a função `round`:[1]

```python
round(10 / 3, 2)
```

```
3.33
```

[1] Você pode aprender mais sobre a função round na documentação oficial do Python.

3.4.3 Strings (str)

Strings são sequências ordenadas de caracteres. No Python, qualquer valor delimitado por aspas simples ou aspas duplas é considerado um string.

```
nome = "Arthur"
print(nome)

curso = 'Economia'
print(curso)
```

```
Arthur
Economia
```

Podemos ainda utilizar uma sequência de três aspas duplas e escrever strings que percorram várias linhas. Isso é bastante útil na documentação de funções personalizadas, como veremos futuramente.

```
texto = """
Das Utopias

Se as coisas são inatingíveis...ora!
Não é motivo para não querê-las...
Que tristes os caminhos, se não fora
A presença distante das estrelas!

Mario Quintana
"""

print(texto)
print(type(texto))
```

```

Das Utopias

Se as coisas são inatingíveis...ora!
Não é motivo para não querê-las...
Que tristes os caminhos, se não fora
A presença distante das estrelas!

Mario Quintana

<class 'str'>
```

3.4.3.1 Formatação e o Problemas das Aspas

Observe como adicionar aspas como parte da sua string é um desafio. `print("Olá, "amigo"")` não funcionará e o interpretador retornará um erro. Geralmente, existem duas abordagens para corrigir isso.

Primeiro, você poderia simplesmente mudar as aspas para aspas simples.

```
print('Olá, "amigo"')
```

```
Olá, "amigo"
```

Uma abordagem mais comum consiste em utilizar o caractere de escape. Por exemplo:

```
print("Olá, \"amigo\"")
```

```
Olá, "amigo"
```

Nesse caso, a barra invertida (\) informa ao interpretador que o caractere imediatamente seguinte deve ser interpretado como parte literal da string — e não como um delimitador da string. Sem o escape, o Python entenderia a segunda aspa dupla como o encerramento da string, produzindo um erro de sintaxe.

O símbolo \ é chamado de caractere de escape (*escape character*). Ele tem duas funções principais:

1. Permitir a inserção de caracteres especiais dentro de strings delimitadas pelo mesmo tipo de aspas (como `\"` ou `\'`);
2. Representar caracteres de controle, isto é, símbolos que não são visíveis, mas produzem efeitos específicos na saída.

Entre os caracteres de escape mais comuns estão:

- `\n` (nova linha),
- `\t` (tabulação),
- `\r` (retorno de linha),
- `\b` (backspace).
- `\\` (barra invertida)

Em termos gerais, o caractere de escape sinaliza ao interpretador que o próximo símbolo deve receber um tratamento especial, seja para ser exibido literalmente, seja para produzir um efeito específico na formatação do texto.

Uma maneira mais elegante e moderna de trabalhar com strings é por meio das `f-strings` (*formatted string literals*), introduzidas no Python 3.6.

```
nome = "Arthur"
print(f"Olá, {nome}")
```

```
Olá, Arthur
```

Observe o `f` antes das aspas em `f"Olá, {nome}"`. Esse prefixo informa ao Python que se trata de uma string formatada. Isso permite inserir variáveis ou expressões diretamente dentro da string, entre chaves `{}`, tornando o código mais legível e natural. Em vez de concatenar:

```
print("Olá, " + nome)
```

```
Olá, Arthur
```

ou usar múltiplos argumentos:

```
print("Olá,", nome)
```

```
Olá, Arthur
```

a f-string permite escrever a mensagem praticamente como ela será exibida. As f-strings também permitem controlar a forma como números são exibidos.

```
valor = 20
raiz_quadrada = 20 ** 0.5
print(f"A raiz quadrada de {valor:d} é {raiz_quadrada:.2f}")
```

```
A raiz quadrada de 20 é 4.47
```

O comando `:d` indica que o valor deve ser tratado como inteiro (`int`). Já o comando `:.2f` indica que o valor deve ser exibido como número de ponto flutuante (`float`) com duas casas decimais.

3.4.3.2 Fatiamento de Strings (Slicing)

O fatiamento de string (*slicing*) permite que você "corte" uma string para obter uma subseção dela. A sintaxe básica utiliza colchetes e índices: `string[início:fim]`.

- Índice de Início: Onde o fatiamento começa (incluído).
- Índice de Fim: Onde o fatiamento termina (excluído).

Assim como outras operações de sequência no Python, o fatiamento (*slicing*) utiliza indexação baseada em zero, onde o primeiro elemento está no índice 0. Pense no índice não como o número do caractere, mas como a distância (ou deslocamento) do início da string.

- No índice 0, você percorreu 0 caracteres (está no início).
- No índice 1, você saltou 1 caractere para chegar ao próximo.

Dica: Se uma string tem comprimento n, o último índice será sempre n−1. Imagine a string `nome = "Python"`. Os índices são:

Letra	P	y	t	h	o	n
Índice	0	1	2	3	4	5
Negativo	-6	-5	-4	-3	-2	-1

```
nome = "Python"

# Pega do índice 0 até o 2 (o 2 não entra)
print(nome[0:2])  # Saída: Py

# Se omitir o início, ele começa do zero
print(nome[:4])   # Saída: Pyth

# Se omitir o fim, ele vai até o último caractere
print(nome[2:])   # Saída: thon

# Pega os três últimos caracteres
print(nome[-3:])     # Saída: hon
# Pega todos caracteres exceto os dois últimos
print(nome[:-2])     # Saída: Pyth
```

Você também pode adicionar um terceiro parâmetro: `string[início:fim:passo]`. O passo determina quantos caracteres o Python deve pular. O Python também permite contar de trás para frente usando números negativos. Note que o -1 é sempre o último caractere.

```
telefone = "0123456789"

# Pega do 1 ao fim, pulando de 2 em 2
print(telefone[1:10:2])  # Saída: 13579

# Pega do início ao fim, pulando de 2 em 2
print(telefone[::2])  # Saída: 02468

# Um truque famoso: inverter uma string
print(nome[::-1])    # Saída: nohtyP
```

3.4.3.3 Métodos de Strings

As strings ilustram bem o conceito de objeto em Python, pois elas vêm acompanhadas de diversos métodos que permitem manipulá-las. Um método pode ser entendido como uma função que pertence a um objeto. Em termos práticos, trata-se de uma sequência de instruções encapsulada em um único comando, que pode receber argumentos e retorna um novo valor. A diferença fundamental é que o método é chamado a partir do próprio objeto, utilizando a notação com ponto: `objeto.metodo()`.

No caso das strings, o Python já oferece, de forma nativa, vários métodos úteis para tratamento e limpeza de texto — algo essencial quando trabalhamos com entrada de dados fornecida pelo usuário. Por exemplo, é comum que o usuário digite espaços antes ou depois do nome. O método `strip()` remove automaticamente quaisquer espaços em branco à esquerda e à direita da string:

```
nome = "   Arthur       "
print(f"Olá, {nome}")

nome = nome.strip()
print(f"Olá, {nome}")
```

```
Olá,    Arthur
Olá, Arthur
```

Após essa operação, mesmo que o usuário digite ” Arthur “, a variável nome armazenará apenas”Arthur”.

Observe que o método retorna uma nova string — ele não altera a string original diretamente. Esse comportamento é consequência do fato de que strings são **imutáveis** em Python.

Podemos ir além da simples remoção de espaços. Muitas vezes desejamos padronizar a forma como o texto é exibido, especialmente quando lidamos com nomes próprios ou títulos. Por exemplo, o método `title()` transforma a string para o chamado formato de título, colocando a primeira letra de cada palavra em maiúscula:

```
curso = "métodos computacionais em economia"

print(curso)
print(curso.title())
```

```
métodos computacionais em economia
Métodos Computacionais Em Economia
```

Já os métodos `lower()` e `upper()` convertem todos os caracteres para minúsculas e maiúsculas, respectivamente.

```
curso = "métodos computacionais em economia"

print(curso.lower())
print(curso.upper())
```

```
métodos computacionais em economia
MÉTODOS COMPUTACIONAIS EM ECONOMIA
```

Para dividir uma string em partes menores — chamadas substrings — com base em um padrão delimitador (como espaço, vírgula ou outro caractere), utilizamos o método `split()`. Esse método "quebra" a string sempre que encontra o delimitador especificado e retorna uma lista contendo os trechos resultantes.

No exemplo a seguir, o espaço `" "` é utilizado como critério de separação. Sempre que o Python encontra um espaço, ele divide o texto.

```
curso = "métodos computacionais em economia"

print(curso.split(" "))
```

```
['métodos', 'computacionais', 'em', 'economia']
```

O método `replace()` permite substituir ocorrências de um trecho de texto por outro dentro de uma string. Sua forma geral é: `string.replace(antigo, novo[, quantidade])`.

- antigo: trecho que será substituído;
- novo: texto que será inserido no lugar;
- quantidade (opcional): número máximo de substituições a serem feitas.

```
texto = "Eu gosto de Java"

print(texto) # Saída: Eu gosto de Java
print(texto.replace("Java", "Python"))  # Saída: Eu gosto de Python
```

```
Eu gosto de Java
Eu gosto de Python
```

Observe dois pontos importantes:

1. A string original (texto) não é alterada.
2. O método retorna uma nova string, pois strings são **imutáveis** em Python.

Podemos controlar quantas ocorrências serão substituídas:

```
frase = "ha ha ha"
print(frase.replace("ha", "he", 2))
```

```
he he ha
```

Você pode aprender mais sobre strings e seus métodos na documentação oficial do Python sobre o tipo str

3.4.4 Booleanos (bool)

O tipo `bool` é utilizado para representar valores lógicos, isto é, valores que podem assumir apenas dois estados possíveis: `True` (Verdadeiro) ou `False` (Falso). Em Python, valores booleanos surgem com frequência como resultado de comparações ou expressões lógicas:

```
print(5 > 3)
print(8 < 2)
```

```
True
False
```

Esses valores serão fundamentais quando estudarmos estruturas de controle, como instruções condicionais (`if`, `elif`, `else`) e estruturas de repetição (`while`, `for`). Em programação, decisões dependem justamente de expressões que avaliam como verdadeiras ou falsas.

Algumas características importantes dos booleanos:

1. Não levam aspas: "True" é uma string, não um booleano.
2. Devem sempre começar com letra maiúscula (`True`, `False`).
3. São um subtipo de inteiro em Python. Isso significa que: `True` equivale a numericamente a 1 e `False` equivale numericamente a 0. Essa característica pode ser útil em operações matemáticas e contagens:

```
resultado = True + True + False + True
print(resultado)
```

```
3
```

3.4.5 Listas

As listas são uma das estruturas de dados mais versáteis do Python. Elas são sequências ordenadas e **mutáveis** que podem armazenar elementos de qualquer tipo: números, strings, objetos ou até outras listas.

A forma mais comum de criar uma lista é colocar os elementos entre colchetes (`[ ]`). Um ponto importante é que listas: um inteiro, uma string, um float e até mesmo outra lista.

Uma lista dentro de outra lista é chamada de lista **aninhada**. Uma lista que não contém elementos é chamada de lista vazia.

```
numeros = [42, 123, 25]
cursos = ["Economia", "Administração", "Contabilidade", "Atuária"]
lista_mista = [42, "Economia", 3.14, [10, 20]] # Contém uma lista aninhada
lista_vazia = []
```

A função `len` pode ser usada para verificar o tamanho de uma lista, ou seja, quantos elementos ela contém. Note que a função `len` conta apenas os elementos da lista, não o número total de itens dentro de listas aninhadas:

```
print(len(numeros))
print(len(cursos))
print(len(lista_mista))
print(len(lista_vazia))
```

```
3
4
4
0
```

3.4.5.1 Operações com Listas

As listas suportam diversas operações, como: concatenação, repetição, verificação de existência de elementos, soma, obtenção do maior e menor valor, entre outras.

```
# Concatenação de listas
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
print(lista1 + lista2)

# Repetição de listas
print(lista1 * 3)

# Verificar se um elemento está ou não na lista
print("Economia" in cursos)
print("Direito" in cursos)
print("Matemática" not in cursos)

# Somar os elementos de uma lista de números
sum(numeros)

# Retornar o maior e o menor valor de uma lista de números
print(max(numeros))
print(min(numeros))
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
True
False
True
123
25
```

3.4.5.2 Fatiamento de Listas (Slicing)

Assim como strings, as listas também suportam o fatiamento (*slicing*), permitindo extrair uma subsequência de elementos. A sintaxe é a mesma: `lista[início:fim:passo]`.

```
print(cursos[1:3])
print(cursos[:2])
print(cursos[2:])
print(cursos[-2:])

numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numeros[::2])
print(numeros[1::2])
print(numeros[::-1])
```

```
['Administração', 'Contabilidade']
['Economia', 'Administração']
['Contabilidade', 'Atuária']
['Contabilidade', 'Atuária']
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

O acesso a listas aninhadas é feito usando colchetes várias vezes:

```
lista_aninhada = [1, 2, [10, 20, 30]]

print(lista_aninhada[2])
print(lista_aninhada[2][0])
print(lista_aninhada[2][1])
print(lista_aninhada[2][-1])
```

```
[10, 20, 30]
10
20
30
```

3.4.5.3 Mutabilidade das listas

Em Python, objetos podem ser **mutáveis** ou **imutáveis**: - **Mutáveis**: podem ser alterados após a criação, sem criar um novo objeto (exemplo: listas). - **Imutáveis**: não podem ser alterados após a criação; qualquer "mudança" cria um novo objeto (exemplo: strings, inteiros, floats).

Strings são objetos **imutáveis**: não é possível modificar seus caracteres diretamente. Qualquer alteração resulta em uma nova string. Por exemplo, tentar modificar o primeiro caractere de uma string gera um erro:

```
string = "economia"
string[0] = "E"
```

```
TypeError: 'str' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[31], line 2
      1 string = "economia"
----> 2 string[0] = "E"
TypeError: 'str' object does not support item assignment
```

Ao contrário das strings, **listas são mutáveis**, ou seja, seus elementos podem ser alterados diretamente, sem criar um novo objeto. No exemplo abaixo, modificamos os elementos da lista `disciplinas` de forma direta — a lista permanece o mesmo objeto na memória, mas seu conteúdo é atualizado:

```
disciplinas = ["micro", "macro", "econometria", "estatística"]

print(disciplinas)

disciplinas[0] = "Microeconomia"
disciplinas[1] = "Macroeconomia"

print(disciplinas)
```

```
['micro', 'macro', 'econometria', 'estatística']
['Microeconomia', 'Macroeconomia', 'econometria', 'estatística']
```

Note que, ao copiar uma lista, ambas as variáveis passam a apontar para o mesmo objeto na memória. Como listas são mutáveis, qualquer alteração feita em uma delas afeta também a outra, pois ambas compartilham a mesma referência.

```python
lista_original = [1, 2, 3]
lista_copia = lista_original

lista_original[1] = 42

print(lista_original)
print(lista_copia)
```

```
[1, 42, 3]
[1, 42, 3]
```

Esse comportamento não ocorre com tipos imutáveis, como inteiros (`int`). Ao copiar um inteiro para outra variável, cada uma passa a referenciar um valor independente. Alterar uma delas não afeta a outra, pois um novo objeto é criado na memória. Por exemplo:

```python
a = 10
b = a
a = 20
print(a)  # 20
print(b)  # 10
```

```
20
10
```

3.4.5.4 Métodos de Listas

Listas possuem diversos **métodos** que permitem adicionar, remover, reorganizar e buscar elementos de forma eficiente. Os principais métodos são: `append`, `extend`, `remove`, `insert`, `reverse` e `index`.

Vamos ilustrar esses métodos com um exemplo divertido: criar uma lista com os resultados possíveis para uma corrida de Mario Kart!

Figura 3.1: Mario Kart

```
resultados = ["Mario", "Luigi"]

# O método append adiciona um novo elemento ao final da lista
resultados.append("Princesa Peach")
resultados.append("Yoshi")
resultados.append("Koopa Troopa")
resultados.append("Toad")

print(resultados)

# Podemos adicionar mais de um elemento por vez
resultados.append(["Bowser", "Donkey Kong"])

# O método remove busca na lista e remove a primeira ocorrência do elemento
resultados.remove(["Bowser", "Donkey Kong"])
```

```
# O método extend adiciona cada elemento de uma lista ao final de outra
resultados.extend(["Bowser", "Donkey Kong"])

print(resultados)
resultados.remove("Bowser")

# O método insert adiciona um elemento em uma posição específica da lista
print(resultados)
resultados.insert(0, "Bowser")

print(resultados)

# O método index retorna o índice da primeira ocorrência do elemento especificado
print(resultados.index("Mario"))

# O método reverse inverte a ordem dos elementos da lista
resultados.reverse()
print(resultados)
```

```
['Mario', 'Luigi', 'Princesa Peach', 'Yoshi', 'Koopa Troopa', 'Toad']
['Mario', 'Luigi', 'Princesa Peach', 'Yoshi', 'Koopa Troopa', 'Toad', 'Bowser', 'Donkey Kong
['Mario', 'Luigi', 'Princesa Peach', 'Yoshi', 'Koopa Troopa', 'Toad', 'Donkey Kong']
['Bowser', 'Mario', 'Luigi', 'Princesa Peach', 'Yoshi', 'Koopa Troopa', 'Toad', 'Donkey Kong
1
['Donkey Kong', 'Toad', 'Koopa Troopa', 'Yoshi', 'Princesa Peach', 'Luigi', 'Mario', 'Bowser
```

3.4.6 Tuplas

Tuplas são sequências ordenadas de elementos, semelhantes às listas, mas com uma diferença fundamental: **tuplas são imutáveis**. Isso significa que, após sua criação, não é possível alterar, adicionar ou remover elementos de uma tupla. Essa característica torna as tuplas úteis para representar dados que não devem ser modificados, como coordenadas, datas ou pares de valores.

Para criar uma tupla, basta separar os valores por vírgula, com ou sem parênteses. Podemos, ainda, criar tuplas usando a função `tuple()`, que converte outros tipos de dados em tuplas.:

```
# Tupla de três números
valores = (10, 20, 30)
print(type(valores))
```

```python
# Tupla de strings
cidades = "São Paulo", "Belo Horizonte", "Recife"
print(type(cidades))

# Tupla mista
dados = ("Economia", 2026, 3.14)

t = tuple()
print(t)

t = tuple('economia')
print(t)

t2 = tuple([1, 2, 3])
print(t2)

t3 = tuple((4, 5, 6))
print(t3)
```

```
<class 'tuple'>
<class 'tuple'>
()
('e', 'c', 'o', 'n', 'o', 'm', 'i', 'a')
(1, 2, 3)
(4, 5, 6)
```

Cuidado: colocar um valor entre parênteses **não** cria uma tupla. Para criar uma tupla com um único elemento, é necessário incluir uma vírgula após o elemento, com ou sem parênteses. Isso indica ao Python que se trata de uma tupla. Essa sintaxe é importante para evitar erros ao manipular dados, especialmente em situações de desempacotamento ou funções que retornam tuplas.

```python
t1 = ('a')
print(type(t1))

t2 = 'a',
print(type(t2))

t3 = ('a',)
print(type(t3))
```

```
<class 'str'>
<class 'tuple'>
<class 'tuple'>
```

3.4.6.1 Operações com Tuplas

A maioria dos operadores de listas também funciona com tuplas. Por exemplo, podemos aplicar as seguintes operações: - indexação (`t[i]`); - fatiamento (`t[i:j:k]`); - obtenção de comprimento (`len(t)`); - concatenação (`t1 + t2`); - replicação (`t * n`) - obtenção do valor mínimo (`min(t)`) e máximo (`max(t)`)

```
t = tuple('economia')

# Indexação: o operador colchetes ([]) acessa um elemento
print(t[0])

# Fatiamento: operador slice seleciona uma parte da tupla
print(t[1:4])

# Número de elementos na tupla
print(len(t))

# Concatenação: o operador + contatena tuplas
print(t + ('F', 'E', 'A'))

# Repetição: o operador * repete a tupla
print(t * 2)

# Valor mínimo e máximo funciona para strings (ordem alfabética)
print(min(t))
print(max(t))

# Para tuplas numéricas:
numeros = (42, 123, 25)
print(min(numeros))
print(max(numeros))
```

```
e
('c', 'o', 'n')
8
('e', 'c', 'o', 'n', 'o', 'm', 'i', 'a', 'F', 'E', 'A')
```

```
('e', 'c', 'o', 'n', 'o', 'm', 'i', 'a', 'e', 'c', 'o', 'n', 'o', 'm', 'i', 'a')
a
o
25
123
```

Ao contrário das listas, tuplas são **imutáveis**: seus elementos não podem ser alterados após a criação. Qualquer tentativa de modificar uma tupla usando o operador de colchetes resultará em um `TypeError`.

```python
# Tupla imutável
t = (1, 2, 3)
t[0] = 99  # TypeError: 'tuple' object does not support item assignment
```

```
TypeError: 'tuple' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[39], line 3
      1 # Tupla imutável
      2 t = (1, 2, 3)
----> 3 t[0] = 99  # TypeError: 'tuple' object does not support item assignment
TypeError: 'tuple' object does not support item assignment
```

Tuplas também **não** possuem métodos para modificar o conteúdo, como `append` ou `remove`.

```python
t.remove(2) # AttributeError: 'tuple' object has no attribute 'remove'
```

```
AttributeError: 'tuple' object has no attribute 'remove'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[40], line 1
----> 1 t.remove(2) # AttributeError: 'tuple' object has no attribute 'remove'
AttributeError: 'tuple' object has no attribute 'remove'
```

3.4.6.2 Atribuição de Tuplas

A atribuição de tuplas em Python permite distribuir os valores de uma tupla diretamente para múltiplas variáveis, tornando o código mais claro e prático. A ideia básica é atribuir os valores de uma tupla (à direita do operador de atribuição `=`) para uma tupla de variáveis (à esquerda do `=`). É fundamental que o número de variáveis à esquerda seja igual ao número de valores à direita do `=`; caso contrário, ocorre um `ValueError`.

De forma mais geral, se o lado esquerdo de uma atribuição for uma tupla, o lado direito pode ser qualquer tipo de sequência — string, lista ou tupla.

```
# Atribuição de tuplas
a, b = 1, 2
a, b

# Para dividir um endereço de e-mail em um nome de usuário e um domínio
email = 'aluno@usp.br'
usuario, dominio = email.split('@')
usuario, dominio

print("Usuário:", usuario)
print("Domínio:", dominio)
```

```
Usuário: aluno
Domínio: usp.br
```

No exemplo a seguir, ocorre o desempacotamento dos valores da tupla `aluno`, atribuindo cada valor a uma variável correspondente do lado esquerdo. Isso torna o código mais legível, facilita o acesso individual aos dados e evita erros de indexação. Esse recurso é muito útil para funções que retornam múltiplos valores, pois permite capturar todos os resultados de forma elegante e organizada, tornando o código mais claro e fácil de manter.

```
aluno = (1234567, "Adam Smith", 20, "Bacharelado em Ciências Econômicas")
nusp, nome, idade, curso = aluno

print(f"NUSP: {nusp}")
print(f"Nome: {nome}")
print(f"Idade: {idade}")
print(f"Curso: {curso}")
```

```
NUSP: 1234567
Nome: Adam Smith
Idade: 20
Curso: Bacharelado em Ciências Econômicas
```

Atribuição de tuplas é útil quando você deseja trocar os valores de duas variáveis.

```
x = 10
y = 20
print(f"Antes da troca: x = {x}, y = {y}")

x, y = y, x # troca os valores de x e y
print(f"Depois da troca: x = {x}, y = {y}")
```

```
Antes da troca: x = 10, y = 20
Depois da troca: x = 20, y = 10
```

3.4.6.3 Função zip()

A função `zip` é utilizada para combinar duas ou mais sequências (listas, tuplas, etc.) em pares, formando uma nova sequência de tuplas. Se os iteráveis tiverem tamanhos diferentes, o `zip` para quando o mais curto terminar. É ideal para combinar dados ou criar dicionários. Para separar listas agrupadas, use o operador `*`.

```
# Combinando listas
nomes = ["Ana", "Bia", "Caio"]
idades = [25, 30, 22]

dados = zip(nomes, idades)
print(list(dados))

# Separando listas agru[adas]
coordenadas = [(-23.5, -46.6), (-22.9, -43.2), (-21.7, -41.3)]

latitude, longitude = zip(*coordenadas)

print(f"Latitude: {latitude}")
print(f"Longitude: {longitude}")
```

```
[('Ana', 25), ('Bia', 30), ('Caio', 22)]
Latitude: (-23.5, -22.9, -21.7)
Longitude: (-46.6, -43.2, -41.3)
```

3.4.7 Conjuntos

Até aqui, estudamos tipos de dados como `str`, `list` e `tuple`, que representam coleções sequenciais. Nessas estruturas, a ordem dos elementos é definida e cada item pode ser acessado por meio de um índice numérico, começando em zero.

Agora, vamos explorar dois tipos fundamentais da linguagem: o `set` (conjunto) e o `dict` (dicionário). Ambos ampliam as possibilidades de manipulação de dados em Python, permitindo operações mais eficientes e adequadas a diferentes contextos.

Conjuntos (`sets`) são estruturas de dados mutáveis, que armazenam elementos **únicos** e **não ordenados**. Isso significa que:

- Não permitem elementos duplicados;
- Não garantem ordem de armazenamento;
- Não suportam indexação nem slicing.

Em Python, conjuntos são definidos por valores separados por vírgulas dentro de chaves `{}`. No exemplo a seguir, observe que, mesmo repetindo “economia”, o conjunto mantém apenas uma ocorrência.

```
# Criando conjuntos
s = {"economia", "administração", "contabilidade", "economia"}
print(s)  # O segundo "economia" será descartado automaticamente

# Conjunto vazio: DEVE usar set(), pois {} cria um dicionário vazio
vazio = set()
```

```
{'economia', 'contabilidade', 'administração'}
```

3.4.7.1 Operações e Métodos

Embora os conjuntos sejam mutáveis (podemos adicionar e remover elementos), seus itens precisam ser imutáveis (por exemplo: `str`, `int`, `float`, `tuple`). Listas e dicionários não podem ser elementos de um set.

Após a criação de um conjunto, **não** é possível alterar seus elementos. No entanto, é possível adicionar novos elementos usando o método `add()`. Note que, se o elemento já estiver presente no conjunto, ele não será adicionado novamente.

```
# Adicionar elementos usando o método add()
s.add("atuária")
print(s)

# Se o elemento estiver presente, o método add() não fará nada
s.add("economia")
print(s)
```

```
{'economia', 'contabilidade', 'administração', 'atuária'}
{'economia', 'contabilidade', 'administração', 'atuária'}
```

Para adicionar múltiplos elementos de uma vez, utilizamos o método `update()`.

```
# Adicionar múltiplos elementos usando o método update()
s.update(['EAE', 'EAD', 'EAC'])
print(s)
```

```
{'administração', 'EAC', 'EAD', 'atuária', 'economia', 'EAE', 'contabilidade'}
```

Para remover elementos do conjunto, podemos usar os métodos `remove()` ou `discard()`. A diferença é que `remove()` gera um erro se o elemento não estiver presente, enquanto `discard()` não.

```
# Remover elementos usando o método remove()
s.remove("EAE")
print(s)

# Isso causará um erro, pois "EAE" já foi removido
s.remove("EAE")

# Isso não causará um erro, mesmo se "EAE" não estiver presente
s.discard("EAE")
```

```
{'administração', 'EAC', 'EAD', 'atuária', 'economia', 'contabilidade'}
```

```
KeyError: 'EAE'
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[48], line 6
      3 print(s)
      5 # Isso causará um erro, pois "EAE" já foi removido
----> 6 s.remove("EAE")
      8 # Isso não causará um erro, mesmo se "EAE" não estiver presente
      9 s.discard("EAE")
KeyError: 'EAE'
```

Mesmo não sendo indexados, conjuntos permitem verificação eficiente de existência usando o operador `in`. A operação de busca em conjuntos é muito eficiente, sendo uma das grandes vantagens dessa estrutura.

```
# Verificar se um elemento está presente usando o operador in
print('economia' in s)
print('matemática' in s)
```

```
True
False
```

Os conjuntos oferecem diversos métodos que facilitam operações clássicas da teoria dos conjuntos. Entre as principais operações, destacam-se:

- **União**: `.union()` ou operador `|`.
- **Interseção**: `.intersection()` ou operador `&`.
- **Diferença**: `.difference()` ou operador `-`.
- **Diferença simétrica**: `.symmetric_difference()` ou operador `^`.

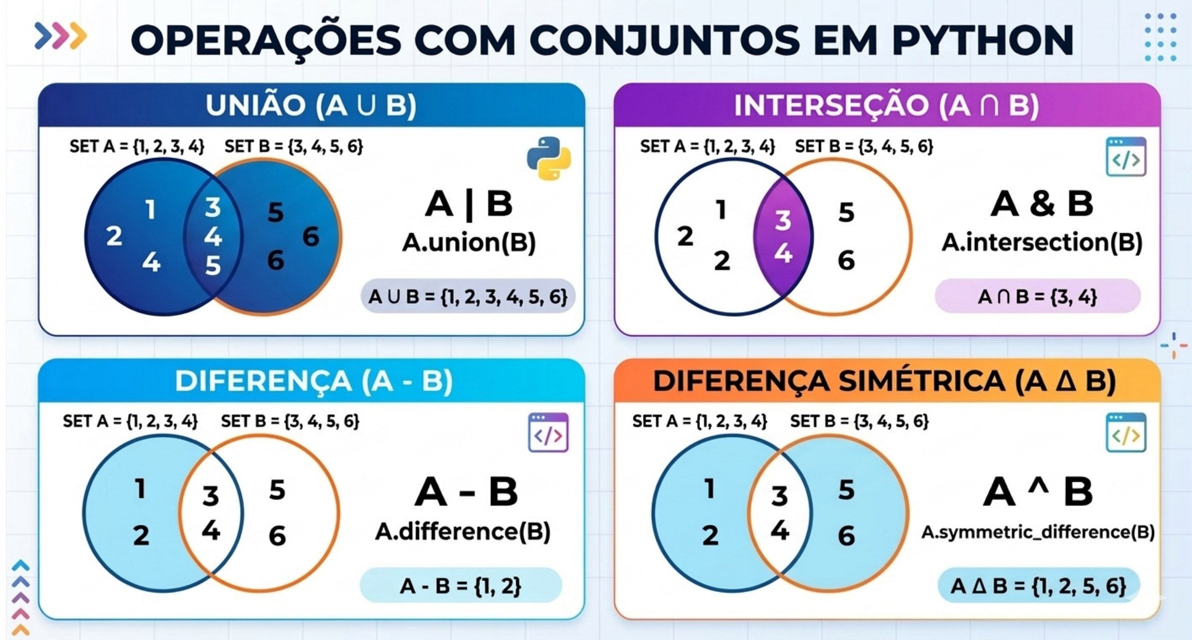


Figura 3.2: Operações com conjuntos

A **união** de dois conjuntos resulta em um novo conjunto contendo todos os elementos presentes em pelo menos um dos conjuntos. Em Python, essa operação pode ser realizada com o método `.union()` ou com o operador `|`.

```python
s1 = {1, 2, 3}
s2 = {3, 4, 5}

# União usando o método union()
s_uniao = s1.union(s2)
print(s_uniao)

# União usando o operador |
s_uniao_operador = s1 | s2
print(s_uniao_operador)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

A **interseção** de dois conjuntos resulta em um novo conjunto contendo apenas os elementos que estão presentes em ambos. Em Python, essa operação pode ser feita com o método `.intersection()` ou com o operador `&`.

```python
# Interseção usando o método intersection()
s_intersecao = s1.intersection(s2)
print(s_intersecao)

# Interseção usando o operador &
s_intersecao_operador = s1 & s2
print(s_intersecao_operador)
```

```
{3}
{3}
```

A **diferença** entre dois conjuntos resulta em um novo conjunto contendo apenas os elementos que estão no primeiro conjunto e não aparecem no segundo. Essa operação é útil para identificar itens exclusivos de um grupo em relação ao outro.

```python
# Diferença usando o método difference()
s_diferenca = s1.difference(s2)
print(s_diferenca)

# Diferença usando o operador -
s_diferenca_operador = s1 - s2
print(s_diferenca_operador)
```

```
{1, 2}
{1, 2}
```

A **diferença simétrica** entre dois conjuntos resulta em um novo conjunto contendo todos os elementos que pertencem a apenas um dos conjuntos, ou seja, exclui os elementos que estão em ambos. É útil para identificar itens exclusivos de cada grupo, eliminando os que são compartilhados.

```
# Diferença simétrica usando o método symmetric_difference()
s_diferenca_simetrica = s1.symmetric_difference(s2)
print(s_diferenca_simetrica)

# Diferença simétrica usando o operador ^
s_diferenca_simetrica_operador = s1 ^ s2
print(s_diferenca_simetrica_operador)
```

```
{1, 2, 4, 5}
{1, 2, 4, 5}
```

Uma das aplicações mais comuns dos conjuntos (`sets`) é a remoção de dados duplicados. Para obter elementos distintos, basta converter a lista original utilizando a função `set()`. Se necessário, o resultado pode ser transformado novamente em lista com `list()`. Esta abordagem é altamente eficiente, embora não garanta a preservação da ordem original dos elementos.

```
nomes = ["Ana", "João", "Ana", "Maria"]
nomes_unicos = list(set(nomes))
print(nomes_unicos)  # Saída: ['Maria', 'Ana', 'João'] (a ordem não é garantida)
```

```
['Maria', 'Ana', 'João']
```

3.4.8 Dicionários

Diferentemente das listas, os dicionários são uma coleção **não sequencial**. Em vez de usar índices numéricos, eles organizam os dados por meio de associações entre **chaves** e **valores**. Assim, podemos pensar em um dicionário como um mapeamento: cada chave aponta para um valor.

- Chave (*key*): identificador único
- Valor (*value*): dado associado à chave

Para dominar os dicionários, precisamos ter em mente três regras:

1. **Unicidade das Chaves**: Cada chave é única. Se você tentar atribuir um novo valor a uma chave já existente, o valor antigo será sobrescrito.
2. **Imutabilidade das Chaves**: As chaves não podem ser alteradas após sua criação. Por isso, devem ser de um tipos imutáveis: `str`, `int`, `float`, `tuple` (desde que a tupla só contenha itens imutáveis). Os tipos `list` ou outros `dict` não podem ser chaves.
3. **Flexibilidade dos Valores**: Os valores podem ser de qualquer tipo, inclusive listas ou outros dicionários.

A forma mais comum de declarar um dicionário é utilizando chaves `{}`. Para declarar um dicionário já preenchido, utilizamos o formato `chave: valor`, separando os pares por vírgulas dentro das chaves `{}`. Também é possível criar dicionários usando a função `dict()`.

```
# Criando um dicionário vazio usando chaves
alunos = {}
print(type(alunos))

# Criando um dicionário com elementos
alunos = {
    'nome': 'Arthur',
    'idade': 20,
    'curso': 'Economia'
}

# Criando um dicionário usando a função dict()
alunos = dict()
print(type(alunos))

# Criando um dicionário com elementos
alunos = dict(
    nome = 'Arthur',
    idade = 20,
    curso = 'Economia'
)

print(alunos)
```

```
<class 'dict'>
<class 'dict'>
{'nome': 'Arthur', 'idade': 20, 'curso': 'Economia'}
```

Outra forma comum de construir dicionários é a partir de uma lista de tuplas, onde cada tupla atua como um par chave-valor. Além disso, podemos associar duas listas distintas utilizando

a função `zip()` em conjunto com `dict()`: a primeira lista fornece as chaves e a segunda, os valores correspondentes.

```
# Usando uma lista de tuplas para criar um dicionário
alunos = dict([
    ('nome', 'Arthur'),
    ('idade', 20),
    ('curso', 'Economia')
])

print(alunos)

# Usando dict() e zip() para criar um dicionário a partir de duas listas
chaves = ['nome', 'idade', 'curso']
valores = ['Arthur', 20, 'Economia']
alunos = dict(zip(chaves, valores))

print(alunos)
```

```
{'nome': 'Arthur', 'idade': 20, 'curso': 'Economia'}
{'nome': 'Arthur', 'idade': 20, 'curso': 'Economia'}
```

3.4.8.1 Acessando e Modificando Itens

Ao contrário das listas e tuplas, que usam índices numéricos para acessar ou modificar elementos (ex: `alunos[0]`), nos dicionários o acesso, criação ou alteração de valores é feito diretamente pela chave, usando colchetes. Quando tentamos acessar o valor de uma chave que **não existe** no dicionário, o Python gera uma exceção do tipo `KeyError`.

```
# Acessar um valor
print(f"Nome: {alunos['nome']}")

# Modificar um valor
alunos["idade"] = 30
print(f"Idade: {alunos['idade']}")

# Adicionar um novo par chave-valor
alunos["disciplina"] = ["Métodos Computacionais", "Microeconomia", "Macroeconomia"]
print(f"Disciplina: {alunos['disciplina']}")

# Acessar um valor inexistente (gera um erro)
print(f"Nome: {alunos['cidade']}")
```

```
Nome: Arthur
Idade: 30
Disciplina: ['Métodos Computacionais', 'Microeconomia', 'Macroeconomia']

KeyError: 'cidade'
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[57], line 13
     10 print(f"Disciplina: {alunos['disciplina']}")
     12 # Acessar um valor inexistente (gera um erro)
---> 13 print(f"Nome: {alunos['cidade']}")
KeyError: 'cidade'
```

Para verificar se uma chave está presente em um dicionário, utilizamos o operador `in`. Por outro lado, para checar se uma chave **não** está presente, usamos o operador `not in`.

```
print('nome' in alunos)
print(f"A chave 'nome' está presente? {'nome' in alunos}")

print('nome' not in alunos)
print(f"A chave 'nome' não está presente? {'nome' not in alunos}")
```

```
True
A chave 'nome' está presente? True
False
A chave 'nome' não está presente? False
```

Para remover uma entrada (par chave-valor) de um dicionário, utilizamos a instrução `del` seguida do nome do dicionário e da chave entre colchetes. Se a chave não existir, será gerado um erro do tipo `KeyError`.

```
# Remover uma entrada do dicionário
del alunos['disciplina']
```

3.4.8.2 Métodos de Dicionários

Os dicionários possuem métodos eficientes para a manipulação e consulta de dados. Embora o acesso via colchetes (`dict[chave]`) seja o mais comum, ele interrompe a execução do código com um `KeyError` caso a chave não seja encontrada. Como alternativa mais robusta, o método `.get()` permite consultar chaves de forma segura, retornando um valor padrão (ou `None`) quando a chave está ausente, o que garante maior estabilidade à aplicação.

```python
# Retorna o valor associado à chave ou valor padrão se a chave não existir
print(f"Nome: {alunos.get('nome')}")

print(f"Cidade: {alunos.get('cidade')}")

print(f"Cidade: {alunos.get('cidade', 'Chave não encontrada')}")
```

```
Nome: Arthur
Cidade: None
Cidade: Chave não encontrada
```

O método `.keys()` retorna uma visão das chaves do dicionário. O resultado é um objeto do tipo `dict_keys`, semelhante a uma lista, contendo todas as chaves presentes no dicionário.

```python
# Retorna uma visão das chaves do dicionário
chaves = alunos.keys()
print(chaves)
print(type(chaves))

# Transformar a visão das chaves em uma lista
lista_chaves = list(alunos.keys())
print(f"Lista de chaves: {lista_chaves}")
```

```
dict_keys(['nome', 'idade', 'curso'])
<class 'dict_keys'>
Lista de chaves: ['nome', 'idade', 'curso']
```

O método `.values()` retorna uma visão dos valores armazenados no dicionário, ou seja, todos os dados associados às suas respectivas chaves. O resultado é um objeto do tipo `dict_values`, semelhante a uma lista, contendo os valores presentes no dicionário.

```python
# Retorna uma visão dos valores do dicionário
valores = alunos.values()
print(valores)
print(type(valores))

lista_valores = list(valores)
print(f"Lista de valores: {lista_valores}")
```

```
dict_values(['Arthur', 30, 'Economia'])
<class 'dict_values'>
Lista de valores: ['Arthur', 30, 'Economia']
```

Já o método `.items()` permite acessar simultaneamente todas as chaves e valores de um dicionário, retornando uma visão composta por tuplas, onde cada tupla contém:

- A chave do dicionário como primeiro elemento;
- O valor associado a essa chave como segundo elemento.

Essa abordagem é útil para percorrer ou manipular pares chave-valor de forma eficiente, especialmente em operações de iteração e processamento de dados.

```python
# Retorna uma visão dos pares chave-valor do dicionário
itens = alunos.items()
print(itens)
print(type(itens))

lista_itens = list(itens)
print(f"Lista de itens: {lista_itens}")
```

```
dict_items([('nome', 'Arthur'), ('idade', 30), ('curso', 'Economia')])
<class 'dict_items'>
Lista de itens: [('nome', 'Arthur'), ('idade', 30), ('curso', 'Economia')]
```

O método `.pop()` remove e retorna o valor da chave especificada:

```python
# Adicionar um novo par chave-valor
alunos['disciplina'] = "Métodos Computacionais"
print(alunos)

# Remove e retorna o valor da chave especificada
alunos.pop('disciplina')
```

```
{'nome': 'Arthur', 'idade': 30, 'curso': 'Economia', 'disciplina': 'Métodos Computacionais'}
```

```
'Métodos Computacionais'
```

O método `.update()` atualiza o dicionário, inserindo ou sobrescrevendo dados:

```python
alunos = {
    'nome': 'Pedro',
    'idade': 20,
    'curso': 'Administração'
}

print(alunos)

# Atualiza o elemento de chave 'nome'
alunos.update({'nome': 'Maria', 'idade': 30, 'curso': 'Economia'})
print(alunos)

# Cria os elementos de chave-valor
alunos.update({'disciplina': ['Métodos Computacionais', 'Microeconomia']})
print(alunos)
```

```
{'nome': 'Pedro', 'idade': 20, 'curso': 'Administração'}
{'nome': 'Maria', 'idade': 30, 'curso': 'Economia'}
{'nome': 'Maria', 'idade': 30, 'curso': 'Economia', 'disciplina': ['Métodos Computacionais',
```

3.4.8.3 Lista de Dicionários

Uma lista de dicionários é uma estrutura de dados robusta, ideal para representar coleções de registros de forma similar a uma tabela de banco de dados. Nesse modelo, a lista atua como a tabela e cada dicionário funciona como uma linha (ou tupla) de dados (ex: `[{'id': 1}, {'id': 2}]`). Embora ofereça a flexibilidade de chaves distintas em cada item, é comumente utilizada para organizar dados homogêneos. Abaixo, veremos como estruturar uma lista de estudantes, onde cada elemento armazena as informações de um aluno:

```python
# Criando uma lista de dicionários
estudantes = [
    {'nome': 'Danilo', 'idade': 20, 'curso': 'Economia'},
    {'nome': 'Arthur', 'idade': 22, 'curso': 'Economia'},
    {'nome': 'Maria', 'idade': 21, 'curso': 'Administração'},
    {'nome': 'João', 'idade': 23, 'curso': 'Contabilidade'},
]

print(estudantes)

# Acessar o terceiro dicionário da lista
print(estudantes[2])
```

```
print(f"Nome: {estudantes[2]['nome']}")
print(f"Idade: {estudantes[2]['idade']}")
print(f"Curso: {estudantes[2]['curso']}")
```

```
[{'nome': 'Danilo', 'idade': 20, 'curso': 'Economia'}, {'nome': 'Arthur', 'idade': 22, 'curs
{'nome': 'Maria', 'idade': 21, 'curso': 'Administração'}
Nome: Maria
Idade: 21
Curso: Administração
```

3.5 Explorando o Python

Programar não significa memorizar todos os métodos e funções da linguagem. Parte essencial do aprendizado é desenvolver autonomia para explorar e descobrir recursos por conta própria. Além da documentação oficial online, o próprio Python oferece ferramentas internas que permitem investigar objetos, tipos e funções diretamente no interpretador.

A função `dir()` funciona como um mapa de possibilidades. Ela lista todos os atributos e métodos disponíveis para um objeto ou tipo de dado. Por exemplo, para descobrir o que pode ser feito com strings:

```
dir(str)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
```

```
'__le__',
'__len__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
```

```
 'maketrans',
 'partition',
 'removeprefix',
 'removesuffix',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Isso responde rapidamente à pergunta: "*Será que existe um método pronto para fazer isso?*". Em vez de procurar manualmente na internet, você pode verificar diretamente no interpretador.

Enquanto `dir()` mostra o que existe, a função `help()` explica como usar. Se você esquecer como funciona o método `strip()`, por exemplo:

```
help(str.strip)
```

```
Help on method_descriptor:

strip(self, chars=None, /)
    Return a copy of the string with leading and trailing whitespace removed.

    If chars is given and not None, remove characters in chars instead.
```

O interpretador exibirá:

- A descrição do método;
- Sua assinatura (quais argumentos aceita);
- O que ele retorna;
- Eventuais observações técnicas.

Isso permite compreender não apenas que um método existe, mas como utilizá-lo corretamente.

3.6 Conclusão

Neste capítulo, exploramos os pilares essenciais do Python: dos tipos básicos de dados (`int`, `float`, `str`, `bool`) às estruturas de dados fundamentais (`lists`, `sets`, `dict`). Mais do que operar e indexar valores, compreendemos a natureza de objeto na linguagem — onde cada dado possui tipo, identidade e métodos próprios. Este fundamento é a base para dominar estruturas complexas e o controle de fluxo. Agora, estamos prontos para avançar rumo às iterações, expandindo significativamente o alcance e a inteligência dos nossos programas.

3.7 Exercícios

1. Considere as duas listas abaixo. Escreva um programa que as converta em um dicionário em que os itens de `keys` correspondam às chaves do dicionários e os itens de `values` os valores.

   ```
   keys = ['Ten', 'Twenty', 'Thirty']
   values = [10, 20, 30]
   ```

2. Considere as atribuições abaixo. Sem executar o código, indique o tipo de cada variável. Explique por que a expressão `a + d` gera um erro. Reescreva-a de forma que a soma aritmética seja realizada corretamente e retorne o valor desejado 20.

   ```
   a=10
   b=3
   c=10.0
   d=1.25
   e='10'
   ```

3. Considere o string `Universidade de São Paulo`. Escreva um código que acesse (i) o primeiro caractere; (ii) o último caractere (sem contar manualmente o tamanho do string); (iii) o nome da cidade utilizando fatiamento.

4. Discorra sobre as principais diferenças entre listas e tuplas. A partir da sua resposta, discuta sobre qual das estruturas é mais indicada para armazenar (i) as notas obtidas por um aluno em determinadas disciplinas e (ii) as coordenadas geográficas de uma cidade (latitude e longitude).

4 Controle de fluxo e iteração

4.1 Controle de Fluxo

Na programação, é muito comum precisarmos que determinados blocos de código sejam executados apenas quando uma condição específica for atendida. Para lidar com esse tipo de situação, o Python oferece mecanismos para controlar o fluxo de execução do programa, chamados de **estruturas condicionais**. As principais estruturas condicionais em Python são: `if`, `elif`, `else`.

Essas estruturas permitem que o programa tome decisões durante a execução, escolhendo diferentes caminhos dependendo das condições definidas. Podemos imaginar isso como uma encruzilhada: dependendo da condição avaliada, o programa pode seguir por um caminho ou por outro. Para que essas decisões sejam possíveis, o Python utiliza **expressões booleanas** e **operadores de comparação e lógicos**, que permitem verificar relações entre valores.

4.1.1 Expressões Booleanas

Uma **expressão booleana** é qualquer expressão que resulta em um dos dois valores lógicos possíveis: `True` (verdadeiro) ou `False` (falso). Essas expressões são essenciais para o controle de fluxo, pois determinam quais blocos de código serão executados.

Expressões booleanas geralmente envolvem operadores de comparação (como `==`, `!=`, `>`, `<`) ou operadores lógicos (`and`, `or`, `not`). Essas expressões podem ser usadas diretamente em estruturas condicionais para tomar decisões no programa.

4.1.1.1 Operadores de Comparação

Os operadores de comparação são usados para comparar dois valores. Esses operadores comparam um valor do lado esquerdo com um valor do lado direito, retornando `True` ou `False`.

Operador	Nome	Função
`==`	Igual a	Verifica se um valor é igual ao outro
`!=`	Diferente de	Verifica se um valor é diferente ao outro

Operador	Nome	Função
`>`	Maior que	Verifica se um valor é maior que outro
`>=`	Maior ou igual	Verifica se um valor é maior ou igual ao outro
`<`	Menor que	Verifica se um valor é menor que outro
`<=`	Menor ou igual	Verifica se um valor é menor ou igual ao outro

4.1.1.2 Operadores Lógicos

Além dos operadores de comparação, Python também possui operadores lógicos que permitem combinar múltiplas condições. Os principais são:

Operador	Definição
`and`	Retorna `True` se ambas as afirmações forem verdadeiras
`or`	Retorna `True` se uma das afirmações for verdadeira
`not`	retorna `False` se o resultado for verdadeiro

4.1.1.3 Operadores de Identidade

Esses operadores são utilizados para comparar objetos, verificando se duas variáveis referenciam o mesmo objeto na memória.

Operador	Definição
`is`	Retorna `True` se ambas as variáveis são o mesmo objeto
`is not`	Retorna `True` se ambas as variáveis não forem o mesmo objeto

4.1.1.4 Operadores de Associação

Esses operadores verificam se um elemento pertence ou não a uma estrutura de dados, como listas, strings ou tuplas.

Operador	Definição
`in`	Retorna `True` caso o valor seja encontrado na sequência
`not in`	Retorna `True` caso o valor não seja encontrado na sequência

4.1.1.5 Operadores de Atribuição

Esses operadores são utilizados para atribuir valores a variáveis e também para atualizar valores existentes.

Operador	Equivalente a
`=`	x = 1
`+=`	x = x + 1
`-=`	x = x - 1
`*=`	x = x * 1
`/=`	x = x / 1
`%=`	x = x % 1

4.1.2 Instruções `if`

Para escrever programas úteis, quase sempre precisamos da capacidade de verificar condições e alterar o comportamento do programa de acordo com elas. As **instruções condicionais** nos dão essa capacidade. A forma mais simples é a instrução `if`. Vamos começar com o seguinte exemplo:

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x < y:
    print("x é menor que y")
```

Observe como o programa recebe a entrada do usuário para x e y, convertendo esses valores para inteiros e armazenando-os nas respectivas variáveis x e y. Em seguida, a instrução `if` compara x e y. Se a condição `x < y` for satisfeita, o comando `print` é executado.

As instruções `if` utilizam valores booleanos (`True` ou `False`) para decidir se um bloco de código deve ou não ser executado. Se a comparação `x < y` for `True`, o interpretador executa o bloco de código indentado.

4.1.3 Controle de Fluxo: `elif` e `else`

Uma instrução `if` pode ter uma segunda parte, chamada de cláusula `else`. A sintaxe é a seguinte:

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x < y:
    print("x é menor que y")
if x > y:
    print("x é maior que y")
if x == y:
    print("x é igual a y")
```

Observe que você está fornecendo uma série de instruções `if`. Primeiro, a primeira instrução `if` é avaliada. Em seguida, a segunda instrução `if` realiza sua avaliação. Por fim, a última instrução `if` também é avaliada. Esse fluxo de decisões é chamado de **controle de fluxo**.

Nosso código pode ser representado da seguinte forma:

Este programa pode ser melhorado evitando fazer três verificações consecutivas. Afinal, nem todas as três condições podem ser verdadeiras ao mesmo tempo!

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x < y:
    print("x é menor que y")
elif x > y:
    print("x é maior que y")
elif x == y:
    print("x é igual a y")
```

A palavra-chave `elif` é uma abreviação para `else if`. Observe como o uso de `elif` permite que o programa faça menos verificações. Primeiro, a instrução `if` é avaliada. Se essa condição for **verdadeira**, todas as instruções `elif` **não serão executadas**.

No entanto, se a instrução `if` for avaliada e considerada **falsa**, o primeiro `elif` será avaliado. Se essa condição for **verdadeira**, a verificação final não será executada. Nosso código pode ser representado da seguinte forma:

Embora o seu computador talvez não perceba diferença de velocidade entre o primeiro programa e essa versão revisada, considere que um servidor online executando bilhões ou trilhões desse tipo de cálculo todos os dias pode ser impactado até mesmo por uma pequena decisão de programação.

Há ainda uma melhoria final que podemos fazer no programa. Observe que, logicamente, a verificação `elif x == y` **não é necessária**. Afinal, se x não é menor que y e x não é maior

que y, então x necessariamente deve ser igual a y. Portanto, não precisamos executar `elif x == y`. Em vez disso, podemos criar um resultado padrão, que será executado caso nenhuma das condições anteriores seja verdadeira, utilizando a instrução `else`. Podemos revisar o código da seguinte forma:

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x < y:
    print("x é menor que y")
elif x > y:
    print("x é maior que y")
else:
    print("x é igual a y")
```

Observe como a complexidade relativa deste programa diminuiu após nossa revisão. Nosso código pode ser representado da seguinte forma:

4.1.4 Operador `or`

O operador `or` permite que o programa tome uma decisão entre uma ou mais alternativas. Por exemplo, podemos modificar ainda mais o nosso programa da seguinte forma:

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x < y or x > y:
    print("x não é igual a y")
else:
    print("x é igual a y")
```

Observe que o resultado do programa permanece o mesmo, mas a complexidade foi reduzida. Com isso, a eficiência do código aumenta. Neste ponto, nosso código já está bastante bom. No entanto, será que o design do programa ainda pode ser melhorado? Podemos modificar nosso código da seguinte forma:

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x != y:
    print("x não é igual a y")
```

```
else:
    print("x é igual a y")
```

Observe que removemos completamente o operador `or` e simplesmente perguntamos: **"x é diferente de y?"**. Assim, fazemos apenas **uma única verificação**, o que torna o código mais eficiente. Para fins de ilustração, também poderíamos modificar nosso código da seguinte forma:

```
x = int(input("Qual o valor de x? "))
y = int(input("Qual o valor de y? "))

if x == y:
    print("x é igual a y")
else:
    print("x não é igual a y")
```

Observe que o operador `==` verifica se os valores à esquerda e à direita são iguais entre si. O uso de dois sinais de igual é muito importante. Se você usar apenas um sinal de igual (`=`), o interpretador provavelmente retornará um erro. Nosso código pode ser ilustrado da seguinte forma:

4.1.5 Operador `and`

Assim como o operador `or`, o operador `and` também pode ser utilizado em instruções condicionais. Comece seu novo programa da seguinte forma:

```
nota = int(input("Nota: "))

if nota >= 90 and nota <= 100:
    print("Conceito: A")
elif nota >=80 and nota < 90:
    print("Conceito: B")
elif nota >=70 and nota < 80:
    print("Conceito: C")
elif nota >=60 and nota < 70:
    print("Conceito: D")
else:
    print("Conceito: F")
```

Observe que, ao executar o programa, você poderá inserir uma nota e receber um conceito. No entanto, perceba que ainda existe potencial para erros (bugs). Normalmente, não devemos

confiar que os usuários sempre irão inserir informações corretas. Podemos melhorar nosso código da seguinte forma:

```
nota = int(input("Nota: "))

if 90 <= nota <= 100:
    print("Conceito: A")
elif 80 <= nota < 90:
    print("Conceito: B")
elif 70 <= nota < 80:
    print("Conceito: C")
elif 60 <= nota < 70:
    print("Conceito: D")
else:
    print("Conceito: F")
```

Observe como o Python permite encadear operadores e condições de uma maneira pouco comum em muitas outras linguagens de programação. Ainda assim, podemos melhorar ainda mais o nosso programa:

```
nota = int(input("Nota: "))

if nota >= 90:
    print("Conceito: A")
elif nota >= 80:
    print("Conceito: B")
elif nota >= 70:
    print("Conceito: C")
elif nota >= 60:
    print("Conceito: D")
else:
    print("Conceito: F")
```

Observe como o programa foi melhorado ao fazer menos verificações. Isso torna o código mais fácil de ler e muito mais fácil de manter no futuro. Você pode aprender mais na documentação oficial do Python sobre controle de fluxo.

4.1.6 Instrução `match`

Assim como as instruções `if`, `elif` e `else`, as instruções `match` também podem ser usadas para executar código condicionalmente, dependendo de determinados valores. Considere o seguinte programa:

```
curso = input("Qual é o seu curso? ")

if curso == "Economia":
    print("FEA")
elif curso == "Administração":
    print("FEA")
elif curso == "Contabilidade":
    print("FEA")
elif curso == "Matemática":
    print("IME")
else:
    print("Curso não encontrado")
```

Observe que as três primeiras instruções condicionais exibem a mesma resposta. Podemos melhorar um pouco esse código utilizando a palavra-chave `or`:

```
curso = input("Qual é o seu curso? ")

if curso == "Economia" or curso == "Administração" or curso == "Contabilidade":
    print("FEA")
elif curso == "Matemática":
    print("IME")
else:
    print("Curso não encontrado")
```

Observe que o número de instruções `elif` diminuiu, melhorando a legibilidade do nosso código. Como alternativa, podemos usar instruções `match` para associar os cursos às unidades da USP. Considere o seguinte código:

```
curso = input("Qual é o seu curso? ")

match curso:
    case "Economia":
        print("FEA")
    case "Administração":
        print("FEA")
    case "Contabilidade":
        print("FEA")
    case "Matemática":
        print("IME")
    case _:
        print("Curso não encontrado")
```

Observe o uso do símbolo `_` no último `case`. Ele corresponde a **qualquer entrada**, resultando em um comportamento semelhante ao de uma instrução `else`. Uma instrução `match` compara o valor que aparece após a palavra-chave `match` com cada um dos valores definidos após as palavras-chave `case`. Quando uma correspondência é encontrada, o bloco de código indentado correspondente é executado e o programa interrompe as demais verificações. Podemos melhorar o código da seguinte forma:

```
curso = input("Qual é o seu curso? ")

match curso:
    case "Economia" | "Administração" | "Contabilidade":
        print("FEA")
    case "Matemática":
        print("IME")
    case _:
        print("Curso não encontrado")
```

Observe o uso da barra vertical simples (`|`). Assim como a palavra-chave `or`, ela permite verificar múltiplos valores dentro da mesma instrução `case`.

4.2 Estruturas de Repetição

Estruturas de repetição permitem executar um mesmo bloco de código múltiplas vezes, tornando o processo mais eficiente e menos sujeito a erros do que repetir comandos manualmente. São fundamentais para automatizar tarefas, processar grandes volumes de dados e implementar algoritmos que dependem de iteração.

Neste capítulo, vamos explorar dois tipos principais de laços (loops):

- **Laços em que não sabemos previamente o número de repetições:** `while()` (executam enquanto uma condição for verdadeira)
- **Laços em que sabemos previamente o número de repetições:** `for()` (executam um número definido de vezes)

Compreender essas estruturas é essencial para resolver problemas de forma eficiente e elegante em programação.

4.2.1 Repetições com `while()`

O princípio de um loop `while` é que as instruções dentro do loop serão repetidas enquanto uma determinada condição for verdadeira. A ideia é que essa condição dependa de um ou

mais objetos (variáveis) que sejam modificados durante as iterações; caso contrário, o loop se tornaria infinito. A sintaxe é a seguinte:

```
while condicao:
    instrucoes
```

Assim como nas instruções condicionais, as instruções devem ser escritas dentro de um **bloco indentado**. Vamos analisar um exemplo de um loop `while`:

```
i = 3

while i != 0:
    print("FEA-USP!")
    i = i - 1
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

Observe que o código executa corretamente, reduzindo o valor de n em 1 a cada "iteração" do loop. O termo **iteração** tem um significado especial em programação: refere-se a cada ciclo completo de execução do laço. A primeira iteração é chamada de iteração 0, a segunda é a iteração 1, e assim por diante.

Em programação, normalmente contamos a partir do zero: 0, 1, 2, etc. Compreender o conceito de iteração é fundamental para entender como os laços funcionam e para controlar corretamente o fluxo de repetição em seus programas. Podemos melhorar o nosso código da seguinte maneira:

```
i = 1

while i <= 3:
    print("FEA-USP!")
    i = i + 1
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

Note que, ao escrevermos `i = i + 1`, estamos atribuindo à variável `i` o valor da expressão à direita. No exemplo acima, começamos i em 1, como a maioria das pessoas costuma contar (1, 2, 3…). Se você executar o código acima, verá que ele imprime cinco vezes.

Entretanto, em programação, é considerado uma boa prática começar a contagem pelo zero. Podemos aprimorar nosso código para iniciar a contagem em zero:

```
i = 0

while i < 3:
    print("FEA-USP!")
    i += 1
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

Perceba que, ao alterar o operador para `i < 3`, nosso código passa a funcionar como esperado: começamos a contagem em 0 e o laço é executado três vezes, imprimindo a mensagem três vezes. - Além disso, repare que `i += 1` é uma forma abreviada de escrever `i = i + 1`, facilitando a leitura e escrita do código. - Nosso código, até aqui, pode ser ilustrado da seguinte maneira:

Note que nosso loop conta o valor de i até 3, mas não inclui o 3.

4.2.2 Repetições com `for()`

Quando sabemos de antemão o número de repetições desejadas, o laço `for()` é a escolha ideal. Sua sintaxe é simples:

```
for objeto in valores_possiveis:
    instrucoes
```

Nesse contexto, `objeto` é uma variável local que, a cada iteração, assume um dos valores definidos em `valores_possiveis`. As `instrucoes` são os comandos executados em cada repetição do laço.

O laço `for` é especialmente útil para percorrer listas, sequências ou qualquer coleção de elementos. Por exemplo, podemos reescrever nosso código anterior da seguinte maneira:

```
for i in [0, 1, 2]:
    print("FEA-USP!")
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

Observe como esse código é mais limpo e fácil de entender em comparação ao exemplo anterior com o loop `while`. Aqui, a variável `i` começa em 0, imprime a mensagem, depois recebe 1, imprime novamente, depois recebe 2, imprime mais uma vez e então o laço termina automaticamente.

Essa abordagem é eficiente para listas pequenas, mas imagine se você precisasse repetir a operação para um milhão de vezes! O ideal é criar códigos que funcionem bem para qualquer quantidade de repetições, tornando-os escaláveis e fáceis de manter. Veja como podemos aprimorar nosso código para esses casos:

```
for i in range(0, 3):
    print("FEA-USP!")
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

A função `range()` é extremamente útil para criar sequências numéricas em laços de repetição, tornando o código mais limpo e eficiente. Seus principais argumentos são:

- `start`: (opcional, padrão 0) valor inicial da sequência (**incluído**);
- `stop`: valor final da sequência (**não incluído**);
- `step`: (opcional, padrão 1) intervalo entre os valores.

Por exemplo, ao utilizar `range(3)`, o Python gera automaticamente os valores 0, 1 e 2, permitindo que o laço execute exatamente três vezes sem a necessidade de criar listas manualmente ou controlar variáveis auxiliares.

Podemos tornar nosso código ainda mais limpo e idiomático. Note que, neste caso, nunca utilizamos a variável `i` dentro do corpo do laço, apenas a usamos para controlar o número de repetições. Ou seja, embora o Python precise de `i` para contar as iterações, não precisamos desse valor para nenhuma outra finalidade.

Em Python, quando a variável de iteração não é utilizada no corpo do laço, é uma boa prática representá-la por um sublinhado (`_`). Isso deixa claro para quem lê o código que o valor não será usado. Veja como fica:

```
for _ in range(3):
    print("FEA-USP!")
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

Note que trocar o `i` por `_` não altera em nada o funcionamento do nosso programa. Na verdade, usar o sublinhado deixa explícito para quem lê o código que o valor da variável de controle não será utilizado, tornando o código mais claro e profissional.

Sempre que você precisar apenas repetir uma ação um certo número de vezes, sem se importar com o valor da variável de iteração, prefira usar `_` no lugar de um nome qualquer. Isso é considerado uma boa prática em Python.

Outra alternativa interessante é utilizar a multiplicação de strings para repetir uma mensagem várias vezes de forma ainda mais concisa. Veja o exemplo:

```
print("FEA-USP!" * 3)
```

```
FEA-USP!FEA-USP!FEA-USP!
```

Note que, ao executar esse código, a mensagem será impressa três vezes, mas o resultado será `FEA-USP!FEA-USP!FEA-USP!` tudo na mesma linha. Como poderíamos adicionar uma quebra de linha ao final de cada mensagem?

```
print("FEA-USP!\n" * 3, end = "")
```

```
FEA-USP!
FEA-USP!
FEA-USP!
```

Note que esse código imprime a mensagem três vezes, cada uma em uma linha diferente. Ao adicionar `end=""` e o `\n`, informamos ao interpretador que deve inserir uma quebra de linha ao final de cada mensagem, mas sem adicionar uma linha em branco extra ao final do resultado.

4.2.3 Mais sobre Listas

Considere o exemplo a seguir:

```
alunos = ["Ana", "Pedro", "Marcos"]

print(alunos[0])
print(alunos[1])
print(alunos[2])
```

```
Ana
Pedro
Marcos
```

Observe que temos uma lista chamada alunos, contendo os nomes dos estudantes. Em seguida, imprimimos o aluno que está na posição 0, "Ana". O mesmo é feito para os demais alunos, acessando cada posição da lista individualmente. No entanto, esse método não é prático se a lista for grande ou se não soubermos quantos elementos ela possui.

Assim como ilustramos anteriormente, podemos usar um laço para percorrer toda a lista de forma automática e eficiente. Para isso, podemos utilizar a função `len()` para descobrir o tamanho (quantidade de elementos) da lista alunos

```python
alunos = ["Ana", "Pedro", "Marcos"]

for i in range(len(alunos)):
    print(alunos[i])
```

```
Ana
Pedro
Marcos
```

Outra forma ainda mais simples e legível de percorrer todos os elementos de uma lista é utilizando o próprio nome da variável na estrutura do laço. Veja:

```python
alunos = ["Ana", "Pedro", "Marcos"]

for aluno in alunos:
    print(aluno)
```

```
Ana
Pedro
Marcos
```

Note que, para cada aluno na lista alunos, o código irá imprimir o nome do aluno como esperado. Você pode se perguntar por que não usamos o sublinhado (`_`), como discutido anteriormente. Optamos por não fazer isso porque a variável aluno é utilizada explicitamente no nosso código.

4.2.4 Interrompendo um Laço de Repetição

Ao trabalhar com laços controlados por condição, pode surgir a necessidade de interromper ou pular partes da execução antes que a condição principal deixe de ser satisfeita. Para esses casos, Python oferece mecanismos explícitos de controle de fluxo. As duas principais palavras-chave para modificar o comportamento de um laço são:

- A instrução `continue` informa explicitamente ao Python para ir para a próxima iteração do laço.
- Já a instrução `break` faz com que o Python "quebre" o laço antes de terminar todas as iterações previstas.

Essas ferramentas são úteis para tratar casos especiais, validar entradas ou evitar cálculos desnecessários dentro de um laço.

```
while True:
    n = int(input("Qual o valor de n? "))
    if n < 0:
        continue
    else:
        break
```

No exemplo anterior, continuamos para a próxima iteração do laço quando n é menor que 0 — o que faz com que o usuário seja novamente solicitado a informar "Qual o valor de n?". Se, por outro lado, n for maior ou igual a 0, saímos do laço e o restante do programa é executado normalmente.

Note que a palavra-chave `continue` é redundante neste caso. Podemos melhorar nosso código da seguinte forma:

```
while True:
    n = int(input("Qual o valor de n? "))
    if n > 0:
        break

for _ in range(n):
    print("FEA-USP!")
```

Observe que esse laço `while` irá sempre executar até que n seja maior que 0. Quando n for maior que 0, o laço é interrompido.

4.2.5 Aplicação: Teorema Central do Limite

O Teorema Central do Limite (TCL) é um dos resultados mais importantes da estatística e da teoria das probabilidades. Em termos simples, ele afirma que, sob certas condições, a média de um grande número de variáveis aleatórias independentes e identicamente distribuídas tende a seguir uma distribuição normal, independentemente da distribuição original dessas variáveis.

> **Teorema Central do Limite**
>
> Suponha que estejamos amostrando de uma variável aleatória X com média finita μ e desvio padrão finito σ. Independentemente da distribuição original de X, a distribuição da *média amostral* aproxima-se de uma distribuição normal à medida que o tamanho da amostra aumenta. Nessa caso, a média e o desvio padrão da distribuição das médias amostrais são dados por:
>
> $$\mu_{\bar{X}} = \mu$$
>
> $$\sigma_{\bar{X}} = \frac{\sigma}{\sqrt{N}}$$

Vamos ilustrar esse resultado por meio de um experimento computacional simples:

1. Gerar várias amostras aleatórias de uma distribuição (por exemplo, uniforme);
2. Calcular a média de cada amostra;
3. Repetir esse processo muitas vezes, armazenando as médias obtidas;
4. Analisar o comportamento dessas médias.

Esse experimento permite observar, na prática, como a distribuição das médias amostrais se aproxima de uma normal, mesmo quando os dados originais não seguem esse formato.

Vamos começar sorteando uma única amostra com 5 observações de uma distribuição uniforme entre 0 e 1:

```python
import numpy as np

amostra = np.random.uniform(0, 1, 5)

print(amostra)
print(np.mean(amostra))
```

```
[0.55239038 0.08503893 0.06768018 0.57103098 0.27461267]
0.3101506278933034
```

Se executarmos esse código novamente, obteremos valores diferentes. Isso acontece porque estamos lidando com números aleatórios. Para tornar os resultados reprodutíveis, utilizamos a função `seed` do NumPy, que fixa o ponto de partida do gerador de números aleatórios:

```python
import numpy as np

np.random.seed(19)
amostra = np.random.uniform(0, 1, 5)

print(amostra)
print(np.mean(amostra))
```

```
[0.0975336  0.76124972 0.24693797 0.13813169 0.33144656]
0.3150599084675772
```

Note que a média da amostra gerada dificilmente será exatamente igual a 0,5, que é a média teórica da distribuição uniforme $U(0,1)$. É aqui que entra o Teorema Central do Limite: embora uma única amostra possa apresentar variações, a média das médias amostrais tende a se aproximar da média populacional conforme aumentamos o número de amostras.

Vamos repetir o experimento várias vezes, gerando diversas amostras de tamanho 5 e calculando suas médias:

```python
import numpy as np

np.random.seed(19)   # reprodutibilidade
n_amostras = 10000   # número de repetições
tamanho_amostra = 5  # tamanho de cada amostra

# Lista para armazenar as médias
medias = []

# Gerar as amostras e calcular as médias
for _ in range(n_amostras):
    amostra = np.random.uniform(0, 1, tamanho_amostra)
    medias.append(np.mean(amostra))

# Calcular a média das médias
media_das_medias = np.mean(medias)
print(f"Média das médias: {media_das_medias:.4f}")
```

```
Média das médias: 0.4989
```

Observe que o valor obtido é muito próximo de 0,5, evidenciando o resultado teórico.

Podemos aprofundar a análise variando o número de repetições do experimento:

```
import numpy as np

np.random.seed(19)                             # reprodutibilidade
n_amostras = [1, 10, 100, 1000, 10000, 100000] # número de repetições
tamanho_amostra = 5                            # tamanho de cada amostra

for n in n_amostras:
    medias = []

    for _ in range(n):
        amostra = np.random.uniform(0, 1, tamanho_amostra)
        medias.append(np.mean(amostra))

    print(f"Média das médias: {np.mean(medias):.4f}")
```

```
Média das médias: 0.3151
Média das médias: 0.5157
Média das médias: 0.4762
Média das médias: 0.5011
Média das médias: 0.4992
Média das médias: 0.5002
```

Note que, à medida que o número de amostras aumenta, a média das médias se estabiliza em torno de 0,5.

4.3 Exercícios

1. Escreva um programa que receba a lista de rendas mensais abaixo e classifique cada observação nas seguintes categorias:

- Dados: `rendas = [1200, 3240, 3250, 7100, 12000, 5000, 7500, 1800, 4500]`
- "Baixa renda" caso a renda mensal seja menor do que 2 salários mínimos.
- "Renda média" caso a renda mensal esteja entre 2 e 5 salários mínimos.
- "Alta renda" caso contrário.

O programa deve retornar uma nova lista contendo as classificações na mesma ordem da lista original. Você deve utilizar as instruções `if` e `for`. Considere o salário mínimo de $R\$1.621$ (valor de 2026)

2. Utilize o loop `while` para construir um código que realize uma soma infinita de uma progressão geométrica em que o primeiro termo é igual a 10 e a razão é igual a 0.9.Lembre-se de definir um critério de parada para que o loop não rode para sempre.

3. Escreva um programa que, a partir de uma lista de dados, percorra todos os elementos um a um ignorando valores negativos, interrompendo o loop caso encontre um número maior do que 8, e somando todos os valores processados pelo loop antes da interrupção. Aplique esse programa à lista `numeros = [4,7,-2,9,1,2,-10,6,-5,3,7,-0.5,12,1,3,-2]`.

4. Simule 10.000 lançamentos de um dado justo utilizando a função `randint` do `numpy.random`. Utilize um loop `for` e expressões condicionais para (i) calcular a frequência relativa de cada face e (ii) a média dos valores obtidos dentro da amostra de lançamentos.

5. O último teorema de *Fermat* diz que não há nenhum número inteiro positivo a,b,c tal que

$$a^n + b^n = c^n$$

para quaisquer valores de n maiores do que 2.

Use o que você aprendeu com condicionais, operações aritméticas e instruções print para testar se o teorema se mantém, dada a lista de números inteiros abaixo. Note que ao fim de cada iteração, o programa deve exibir "Holy smokes! Fermat was wrong" caso o teorema não valha e "Fermat was right" caso você não tenha refutado um gênio dos tempos modernos.

- `a = [1,2,3,4,5,6,7,8,9,10]`
- `b = [1,2,3,4,5,6,7,8,9,10]`
- `n = [3,4,5,6,7,8,9,10,37,52,89,100]`

5 Funções

Até aqui, nossos programas seguiam um fluxo linear: o Python executava cada instrução na ordem em que ela aparecia. Essa abordagem funciona bem para tarefas simples, mas rapidamente se torna problemática à medida que os programas crescem.

Imagine que você precisa calcular o PIB pela ótica da despesa em vinte cenários diferentes. Sem funções, você repetiria a mesma fórmula vinte vezes — e bastaria um erro em uma das cópias para comprometer toda a análise. **Funções** resolvem esse problema: elas permitem que um bloco de código seja escrito uma única vez e reutilizado quantas vezes for necessário, com diferentes valores de entrada.

> **Analogia econômica:** pense em uma função como uma **máquina de produção**. Você fornece *insumos* (parâmetros), a máquina executa um *processo* (instruções) e entrega um *produto* (retorno). A mesma máquina pode ser acionada várias vezes com insumos diferentes, sempre produzindo o resultado correto.

5.1 Definindo uma função

Para definir uma função em Python, usamos a palavra-chave `def`, seguida do nome da função e dos parâmetros entre parênteses. Todo o bloco de código indentado abaixo do cabeçalho fará parte da função.

```
def nome_da_funcao(parametros):
    <instruções>
    ...
    return <valor>  # opcional
```

Os elementos da sintaxe são:

- **`def`**: indica ao Python que uma função está sendo definida.
- **`nome_da_funcao`**: nome descritivo que indica o que a função faz.
- **`parametros`**: valores que a função recebe para processar (opcional).
- **`return`**: devolve um resultado para quem chamou a função (opcional).

O exemplo mais simples é uma função sem parâmetros:

```
def saudacao():
    print("Olá, seja bem-vindo(a)!")


saudacao()
```

```
Olá, seja bem-vindo(a)!
```

5.2 Parâmetros

Os **parâmetros** tornam as funções flexíveis: permitem que diferentes valores sejam processados sem alterar o código da função. O mesmo bloco pode ser reutilizado em várias partes do programa com diferentes entradas.

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


saudacao("Arthur")
saudacao("Mariana")
```

```
Olá, Arthur! Seja bem-vindo(a)!
Olá, Mariana! Seja bem-vindo(a)!
```

5.2.1 Valores padrão

Podemos atribuir um **valor padrão** a um parâmetro. Se nenhum argumento for passado, o Python usa o padrão automaticamente:

```
def saudacao(nome="visitante"):
    print(f"Olá, {nome}! Seja bem-vindo(a)!")


saudacao("Arthur")
saudacao()           # usa o padrão: "visitante"
```

```
Olá, Arthur! Seja bem-vindo(a)!
Olá, visitante! Seja bem-vindo(a)!
```

5.2.2 Parâmetros posicionais e nomeados

Quando uma função recebe vários parâmetros, o Python os associa **pela ordem** em que são passados — isso é chamado de **argumento posicional**:

```python
def apresentar(nome, curso):
    print(f"Meu nome é {nome} e o meu curso é {curso}")


apresentar("Arthur", "Economia")
```

```
Meu nome é Arthur e o meu curso é Economia
```

Se a ordem for trocada acidentalmente, o resultado fica semanticamente incorreto:

```python
apresentar("Economia", "Arthur")  # saída incorreta!
```

```
Meu nome é Economia e o meu curso é Arthur
```

Para evitar esse tipo de erro, podemos usar **argumentos nomeados**, informando explicitamente a qual parâmetro cada valor pertence. Com argumentos nomeados, a ordem não importa:

```python
apresentar(nome="Arthur", curso="Economia")
apresentar(curso="Economia", nome="Arthur")  # mesmo resultado
```

```
Meu nome é Arthur e o meu curso é Economia
Meu nome é Arthur e o meu curso é Economia
```

5.2.3 Exemplo: juros compostos

A fórmula de juros compostos $M = C \cdot (1 + i)^t$ é um conceito central em finanças. O parâmetro padrão `periodos=1` é útil quando queremos calcular o rendimento de um único período sem precisar especificá-lo toda vez:

```python
def juros_compostos(capital, taxa, periodos=1):
    montante = capital * (1 + taxa) ** periodos
    rendimento = montante - capital
    print(f"Capital inicial:  R$ {capital:,.2f}")
    print(f"Taxa por período: {taxa*100:.1f}%")
    print(f"Períodos:         {periodos}")
    print(f"Rendimento:       R$ {rendimento:,.2f}")
    print(f"Montante final:   R$ {montante:,.2f}")


print("--- Rendimento da poupança em 1 mês ---")
juros_compostos(capital=10_000, taxa=0.005)        # usa o padrão periodos=1

print()
print("--- Rendimento do Tesouro Selic em 12 meses ---")
juros_compostos(capital=10_000, taxa=0.005, periodos=12)
```

```
--- Rendimento da poupança em 1 mês ---
Capital inicial:  R$ 10,000.00
Taxa por período: 0.5%
Períodos:         1
Rendimento:       R$ 50.00
Montante final:   R$ 10,050.00

--- Rendimento do Tesouro Selic em 12 meses ---
Capital inicial:  R$ 10,000.00
Taxa por período: 0.5%
Períodos:         12
Rendimento:       R$ 616.78
Montante final:   R$ 10,616.78
```

5.3 Retornando valores

Até agora, nossas funções apenas exibiam resultados com `print()`. A palavra-chave `return` permite que uma função **devolva um resultado** para quem a chamou, possibilitando seu uso em expressões e cálculos posteriores.

Essa é uma das distinções mais importantes para iniciantes:

```python
# Versão com print: o resultado é exibido, mas não pode ser reutilizado
def dobro_print(n):
    print(n * 2)

# Versão com return: o resultado é devolvido e pode ser usado em cálculos
def dobro_return(n):
    return n * 2


resultado_print = dobro_print(5)    # exibe 10...
print(resultado_print)              # ...mas a variável guarda None!

print()

resultado_return = dobro_return(5)
print(resultado_return)             # 10
print(resultado_return + 3)         # 13 - podemos continuar operando!
```

```
10
None

10
13
```

`print()` vs. `return`

Use `print()` quando quiser apenas **mostrar** algo ao usuário. Use `return` quando quiser que a função **produza um valor** que será usado em outro lugar do programa. Para construir modelos econômicos com vários cálculos encadeados, `return` é indispensável.

5.3.1 Exemplo: encadeando cálculos

A **modularização** — dividir um problema em funções menores, cada uma fazendo uma coisa só — é uma das práticas mais importantes em programação. O exemplo abaixo calcula o salário líquido descontando INSS e IRPF, com cada etapa em sua própria função:

```python
def calcular_inss(salario_bruto):
    teto_inss = 8_475.55  # teto do INSS em 2026
    salario_inss = min(salario_bruto, teto_inss)
```

```
    faixas_inss = [
        (1_621.00, 0.075),
        (2_902.84, 0.09),
        (4_354.27, 0.12),
        (teto_inss, 0.14)
    ]

    inss = 0
    anterior = 0
    for limite, aliquota in faixas_inss:
        if salario_inss > limite:
            inss += (limite - anterior) * aliquota
            anterior = limite
        else:
            inss += (salario_inss - anterior) * aliquota
            break
    return inss

def calcular_irpf(base_irpf):
    # Tabela mensal do IRPF vigente a partir de maio de 2025
    faixas_irpf = [
        (2_428.80, 0.00, 0.00),
        (2_826.65, 0.075, 182.16),
        (3_751.05, 0.15, 394.16),
        (4_664.68, 0.225, 675.49),
        (float("inf"), 0.275, 908.73),
    ]

    irpf = 0
    for limite, aliquota, deducao in faixas_irpf:
        if base_irpf <= limite:
            irpf = (base_irpf * aliquota) - deducao
            break

    return max(irpf, 0)  # IR não pode ser negativo

def salario_liquido(salario_bruto):
    desconto_inss = calcular_inss(salario_bruto)
    base_irpf = salario_bruto - desconto_inss
    desconto_irpf = calcular_irpf(base_irpf)
    liquido = salario_bruto - desconto_inss - desconto_irpf
    return liquido
```

```python
# Testando para diferentes faixas salariais
salarios = [1_500, 3_000, 6_000, 12_000]

print(f"{'Salário Bruto':>16} | {'INSS':>10} | {'IRPF':>10} | {'Salário Líquido':>16}")
print("-" * 62)

for bruto in salarios:
    inss = calcular_inss(bruto)
    irpf = calcular_irpf(bruto - inss)
    liquido = salario_liquido(bruto)
    print(f"R$ {bruto:>12,.2f} | R$ {inss:>7,.2f} | R$ {irpf:>7,.2f} | R$ {liquido:>13,.2f}")
```

```
   Salário Bruto |       INSS |       IRPF |  Salário Líquido
--------------------------------------------------------------
R$     1,500.00 | R$  112.50 | R$    0.00 | R$      1,387.50
R$     3,000.00 | R$  248.60 | R$   24.20 | R$      2,727.21
R$     6,000.00 | R$  641.51 | R$  564.85 | R$      4,793.63
R$    12,000.00 | R$  988.09 | R$ 2,119.54 | R$      8,892.36
```

5.3.2 Múltiplos retornos

Uma função pode retornar **mais de um valor** separando-os por vírgula. O Python os agrupa em uma **tupla**, que pode ser desempacotada em variáveis distintas:

```python
def operacoes(a, b):
    soma    = a + b
    produto = a * b
    return soma, produto


s, p = operacoes(2, 3)
print(s, p)  # Saída: 5 6
```

```
5 6
```

Também é possível retornar dicionários, o que é especialmente útil quando os resultados têm nomes significativos:

```python
import statistics

def descritivas(x):
    media        = statistics.mean(x)
    desvio_padrao = statistics.stdev(x)
    return {"media": media, "desvio_padrao": desvio_padrao}


dados = [1, 2, 3, 4, 5]
print(descritivas(dados))
```

```
{'media': 3, 'desvio_padrao': 1.5811388300841898}
```

5.3.3 Exemplo: métricas de inflação

O exemplo abaixo calcula a inflação acumulada e a média mensal a partir de uma lista de taxas mensais, retornando ambas em uma única chamada:

```python
def analisar_inflacao(taxas_mensais):
    # Inflação acumulada: produto de (1 + taxa) para cada mês
    acumulada = 1
    for taxa in taxas_mensais:
        acumulada *= (1 + taxa)
    acumulada -= 1  # converte de volta para formato decimal

    media_mensal = statistics.mean(taxas_mensais)
    return acumulada, media_mensal


# IPCA mensal hipotético (em decimal)
ipca_2024 = [0.0042, 0.0083, 0.0016, 0.0038, 0.0044,
             0.0050, 0.0038, 0.0044, 0.0044, 0.0056,
             0.0039, 0.0052]

acum, media = analisar_inflacao(ipca_2024)

print(f"Inflação acumulada no ano: {acum*100:.2f}%")
print(f"Inflação média mensal:     {media*100:.2f}%")
```

```
Inflação acumulada no ano: 5.60%
Inflação média mensal:     0.46%
```

5.4 Aplicação: Teorema Central do Limite

No capítulo anterior, escrevemos um código para simular o Teorema Central do Limite (TCL): ao retirar muitas amostras de uma distribuição e calcular a média de cada uma, a distribuição dessas médias tende a ser normal — independentemente da distribuição original.

O código original funcionava, mas tinha uma limitação: os parâmetros da simulação estavam fixados diretamente no código. Para testar cenários diferentes, era preciso reescrevê-lo toda vez.

Transformar esse código em uma **função reutilizável** resolve o problema. Os parâmetros com valores padrão (`n_amostras=10000`, `tamanho_amostra=30`, `seed=1234`) deixam a função flexível: ela funciona sem nenhum argumento, mas também aceita configurações personalizadas.

```python
import numpy as np

def simular_tcl(n_amostras=10000, tamanho_amostra=30, seed=1234):
    """
    Simula o Teorema Central do Limite usando amostras de uma
    distribuição uniforme entre 0 e 1.

    Parâmetros:
        n_amostras (int): número de amostras a gerar. Padrão: 10000.
        tamanho_amostra (int): observações por amostra. Padrão: 30.
        seed (int): semente para reprodutibilidade. Padrão: 1234.

    Retorna:
        float: média das médias amostrais.
    """
    np.random.seed(seed)
    medias = []

    for _ in range(n_amostras):
        amostra = np.random.uniform(0, 1, tamanho_amostra)
        medias.append(np.mean(amostra))

    return np.mean(medias)


resultado = simular_tcl()
print(f"Média das médias (padrão): {resultado:.4f}")
```

```
Média das médias (padrão): 0.4994
```

5.4.1 Verificando a convergência

Uma das demonstrações clássicas do TCL é observar como a média das médias converge para o valor teórico ($\mu = 0{,}5$ no caso da $U(0,1)$) conforme aumentamos o número de amostras. Com a função, isso fica muito mais claro:

```
lista_n_amostras = [1, 10, 100, 1_000, 10_000, 100_000]

print(f"{'Nº de amostras':>16} | {'Média das médias':>18} | {'Erro vs. 0.5':>14}")
print("-" * 56)

for n in lista_n_amostras:
    media = simular_tcl(n_amostras=n, tamanho_amostra=5)
    erro  = abs(media - 0.5)
    print(f"{n:>16,} | {media:>18.4f} | {erro:>14.4f}")
```

```
  Nº de amostras |   Média das médias |   Erro vs. 0.5
--------------------------------------------------------
               1 |             0.5633 |         0.0633
              10 |             0.5225 |         0.0225
             100 |             0.5205 |         0.0205
           1,000 |             0.4992 |         0.0008
          10,000 |             0.5002 |         0.0002
         100,000 |             0.4996 |         0.0004
```

Note que, ao aumentar o número de amostras, a média das médias se aproxima cada vez mais de 0,5 — exatamente o TCL em ação. E com a função, podemos explorar esse comportamento variando os parâmetros com apenas uma linha de código.

5.5 Docstrings

Docstrings são strings de documentação escritas logo abaixo do cabeçalho de uma função, entre aspas triplas (`"""`). Elas descrevem o que a função faz, quais parâmetros recebe e o que retorna.

Boas docstrings tornam o código muito mais fácil de entender e manter — especialmente quando você retorna a ele depois de semanas, ou quando compartilha com colegas. Você pode consultá-las a qualquer momento com `help()`:

```python
def taxa_desemprego(desempregados, populacao_ativa):
    """
    Calcula a taxa de desemprego de uma economia.

    Parâmetros:
        desempregados (float): número de pessoas desempregadas.
        populacao_ativa (float): tamanho da força de trabalho (PEA).

    Retorna:
        float: taxa de desemprego, expressa em percentual (%).
    """
    return (desempregados / populacao_ativa) * 100


help(taxa_desemprego)
```

```
Help on function taxa_desemprego in module __main__:

taxa_desemprego(desempregados, populacao_ativa)
    Calcula a taxa de desemprego de uma economia.

    Parâmetros:
        desempregados (float): número de pessoas desempregadas.
        populacao_ativa (float): tamanho da força de trabalho (PEA).

    Retorna:
        float: taxa de desemprego, expressa em percentual (%).
```

```python
# Dados hipotéticos (em milhões de pessoas)
taxa = taxa_desemprego(desempregados=8.5, populacao_ativa=107.0)
print(f"Taxa de desemprego: {taxa:.1f}%")
```

```
Taxa de desemprego: 7.9%
```

5.6 Escopo de variáveis

O **escopo** de uma variável se refere ao local onde ela é definida e onde pode ser acessada. Python segue a regra **LEGB** para resolver os nomes: procura primeiro no escopo **L**ocal, depois no **E**nclosing, depois no **G**lobal e por fim no **B**uilt-in.

5.6.1 Escopo local

Uma variável definida dentro de uma função existe apenas dentro dela:

```
def funcao_local():
    mensagem = "Olá, Mundo!"  # variável local
    print(mensagem)


funcao_local()      # funciona
print(mensagem)     # NameError: não existe fora da função
```

```
Olá, Mundo!

NameError: name 'mensagem' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[17], line 7
      3     print(mensagem)
      6 funcao_local()      # funciona
----> 7 print(mensagem)     # NameError: não existe fora da função
NameError: name 'mensagem' is not defined
```

5.6.2 Escopo enclosing

Se uma função está dentro de outra, a interna pode acessar as variáveis da externa — mas não o contrário:

```
def funcao_externa():
    mensagem = "Olá, Mundo!"

    def funcao_interna():
        print(mensagem)   # acessa a variável da função externa

    funcao_interna()


funcao_externa()
```

```
Olá, Mundo!
```

5.6.3 Escopo global

Uma variável definida fora de todas as funções pode ser **lida** de qualquer lugar. No entanto, para **modificá-la** dentro de uma função, é necessário declará-la com a palavra-chave `global`.

Primeiro, a leitura sem problema:

```
poupanca = 1000

def banco():
    print(f"Saldo atual: R$ {poupanca}")   # lê a variável global


banco()
```

```
Saldo atual: R$ 1000
```

Agora, a tentativa de modificação **sem** `global`:

```
poupanca = 1000

def deposito(n):
    poupanca += n    # UnboundLocalError!

deposito(500)
```

```
UnboundLocalError: local variable 'poupanca' referenced before assignment
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[20], line 6
      3 def deposito(n):
      4     poupanca += n    # UnboundLocalError!
----> 6 deposito(500)
Cell In[20], line 4, in deposito(n)
      3 def deposito(n):
----> 4     poupanca += n
UnboundLocalError: local variable 'poupanca' referenced before assignment
```

> **ℹ Por que o `UnboundLocalError`?**
>
> Ao encontrar `poupanca += n`, o Python interpreta `poupanca` como uma variável *local* da função — pois está sendo atribuída. Mas como ela ainda não tem valor local, o Python

não sabe de onde buscá-la.

A solução é usar `global` para avisar ao Python que a variável em questão pertence ao escopo global:

```
poupanca = 1000

def deposito(n):
    global poupanca
    poupanca += n

def retirada(n):
    global poupanca
    poupanca -= n

def banco():
    deposito(500)
    retirada(200)
    print(f"Saldo atual: R$ {poupanca}")


banco()
```

```
Saldo atual: R$ 1300
```

Boa prática: evite `global`

Embora `global` resolva o problema, seu uso excessivo dificulta a leitura e a manutenção do código — fica difícil rastrear onde uma variável está sendo alterada. Em programas maiores, é preferível **passar o valor como parâmetro e retorná-lo explicitamente**. Você aprenderá formas mais robustas de gerenciar estado à medida que avançar no curso.

5.6.4 Escopo enclosing com `nonlocal`

A palavra-chave `nonlocal` funciona de forma similar a `global`, mas atua no escopo *enclosing*: permite que uma função interna modifique uma variável da função que a contém.

No exemplo abaixo, `transacao` precisa de `nonlocal` para incrementar o contador definido em `registrar_transacoes`:

```python
def registrar_transacoes():
    total_transacoes = 0

    def transacao(descricao):
        nonlocal total_transacoes
        total_transacoes += 1
        print(f"  Transação {total_transacoes}: {descricao}")

    transacao("Depósito de R$ 500")
    transacao("Pagamento de boleto R$ 120")
    transacao("Transferência de R$ 200")
    print(f"Total de transações registradas: {total_transacoes}")


registrar_transacoes()
```

```
  Transação 1: Depósito de R$ 500
  Transação 2: Pagamento de boleto R$ 120
  Transação 3: Transferência de R$ 200
Total de transações registradas: 3
```

5.6.5 Escopo built-in

O escopo *built-in* contém os nomes pré-definidos do Python — funções como `print`, `len`, `range` e `sum`. Eles estão sempre disponíveis, em qualquer parte do código:

```python
print(len("Python"))   # len é uma função built-in
```

```
6
```

5.7 args e kwargs

`*args` e `**kwargs` são formas flexíveis de passar múltiplos argumentos para uma função, permitindo lidar com quantidades variáveis de dados — algo bastante útil em modelos econômicos, onde o número de variáveis pode mudar.

Observe, por exemplo, a assinatura da função `print`:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- `*objects` aceita qualquer número de argumentos posicionais: `print("a", "b", "c")`.
- `end='\n'` é um argumento nomeado com valor padrão: `print("texto", end="")`.

> **`args` e `kwargs` são convenções, não palavras reservadas**
>
> O que importa são os asteriscos: `*` coleta múltiplos argumentos posicionais em uma tupla, e `**` coleta múltiplos argumentos nomeados em um dicionário. Você poderia escrever `*numeros` ou `**indicadores` com o mesmo resultado. `args` e `kwargs` são apenas o padrão adotado pela comunidade Python.

5.7.1 `*args`: múltiplos argumentos posicionais

```
def adicao(*args):
    resultado = 0
    for argumento in args:
        resultado += argumento
    return resultado


print(adicao(1, 2))
print(adicao(1, 2, 3, 4))
print(adicao(1, 2, 3, 4, 5, 6))
```

```
3
10
21
```

5.7.1.1 Exemplo: PIB com composição variável

Uma das identidades macroeconômicas mais fundamentais é:

$$Y = C + I + G + (X - M)$$

onde C é o consumo das famílias, I o investimento, G os gastos do governo, X as exportações e M as importações. Em vez de repetir essa fórmula toda vez que precisarmos calculá-la, podemos encapsulá-la em uma função:

```python
def calcular_pib(consumo, investimento, gastos_governo, exportacoes, importacoes):
    pib = consumo + investimento + gastos_governo + (exportacoes - importacoes)
    print(f"PIB calculado: R$ {pib:.2f} bilhões")


# Dados hipotéticos (em bilhões de R$)
calcular_pib(
    consumo=4_800,
    investimento=1_100,
    gastos_governo=2_200,
    exportacoes=1_600,
    importacoes=1_400
)
```

```
PIB calculado: R$ 8300.00 bilhões
```

Com `*args`, podemos criar uma função que soma todos os componentes do PIB, independentemente de quantos forem passados:

```python
def calcular_pib(*componentes):
    """Soma todos os componentes do PIB passados como argumentos."""
    return sum(componentes)


pib_simples  = calcular_pib(4800, 2200, 1600)              # C + I + G
pib_completo = calcular_pib(4800, 1100, 2200, 1600, -1400) # C+ I + G + (X - M)

print(f"PIB simplificado: R$ {pib_simples:,.0f} bi")
print(f"PIB completo:     R$ {pib_completo:,.0f} bi")
```

```
PIB simplificado: R$ 8,600 bi
PIB completo:     R$ 8,300 bi
```

5.7.2 `**kwargs`: múltiplos argumentos nomeados

`**kwargs` agrupa os argumentos passados com nome em um dicionário dentro da função:

```python
def concatenar(**kwargs):
    print(f"Valores recebidos: {kwargs}")
    resultado = ""
    for valor in kwargs.values():
        resultado += f"{valor} "
    return resultado


print(concatenar(curso="Economia"))
print(concatenar(curso="Economia", faculdade="FEA-USP"))
```

```
Valores recebidos: {'curso': 'Economia'}
Economia
Valores recebidos: {'curso': 'Economia', 'faculdade': 'FEA-USP'}
Economia FEA-USP
```

5.7.2.1 Exemplo: relatório de indicadores macroeconômicos

Com `**kwargs`, podemos gerar relatórios com qualquer conjunto de variáveis — útil quando comparamos países com disponibilidades de dados diferentes:

```python
def relatorio_macro(pais, **indicadores):
    """
    Exibe um relatório de indicadores macroeconômicos.

    Parâmetros:
        pais (str): nome do país.
        **indicadores: pares nome=valor para qualquer indicador econômico.
    """
    print(f"\n=== Indicadores Macroeconômicos: {pais} ===")
    for indicador, valor in indicadores.items():
        nome_formatado = indicador.replace("_", " ").title()
        print(f"  {nome_formatado}: {valor}")


relatorio_macro(
    "Brasil",
    pib_crescimento="2.9%",
    inflacao_ipca="4.83%",
    taxa_desemprego="6.2%",
    taxa_selic="10.5%"
```

```
)

relatorio_macro(
    "Argentina",
    inflacao="211%",
    taxa_desemprego="7.7%"
)
```

```
=== Indicadores Macroeconômicos: Brasil ===
  Pib Crescimento: 2.9%
  Inflacao Ipca: 4.83%
  Taxa Desemprego: 6.2%
  Taxa Selic: 10.5%

=== Indicadores Macroeconômicos: Argentina ===
  Inflacao: 211%
  Taxa Desemprego: 7.7%
```

5.7.3 Combinando `*args` e `**kwargs`

```
def relatorio_economico(*args, **kwargs):
    print("Valores agregados:", sum(args))
    for chave, valor in kwargs.items():
        print(f"{chave}: {valor}")


relatorio_economico(100, 200, consumo=5_000, investimento=2_000)
```

```
Valores agregados: 300
consumo: 5000
investimento: 2000
```

5.8 Funções anônimas (lambda)

Funções **lambda** são funções anônimas — definidas sem nome, em uma única linha. São úteis para operações simples e temporárias, especialmente como argumento para outras funções. A sintaxe é:

```
lambda <argumentos> : <expressão>
```

Um exemplo simples:

```
soma = lambda x, y: x + y
print(soma(5, 3))
```

```
8
```

5.8.1 Função `map()`

`map()` aplica uma função a cada elemento de uma sequência. Com lambda, isso fica muito conciso:

```
numeros  = [1, 2, 3, 4, 5]
quadrados = list(map(lambda x: x ** 2, numeros))
print(quadrados)
```

```
[1, 4, 9, 16, 25]
```

5.8.1.1 Exemplo: corrigindo preços pela inflação

Em economia, **deflacionar** uma série de preços significa remover o efeito da inflação para tornar valores de diferentes períodos comparáveis em termos reais. A fórmula geral é:

$$P_{\text{real}} = \frac{P_{\text{nominal}}}{1 + \pi}$$

onde P_{nominal} é o preço observado no período e π é a taxa de inflação acumulada entre o período de referência e o período de base. O denominador $(1 + \pi)$ atua como um **deflator**: quanto maior a inflação, mais o preço nominal é "encolhido" para revelar seu valor real.

> **Valor nominal vs. valor real**
>
> Um **valor nominal** é expresso nos preços correntes do período em que foi medido. Um **valor real** é corrigido pela inflação e expressa o poder de compra em termos de um período de referência (base). A distinção é fundamental em macroeconomia: o PIB pode crescer em termos nominais simplesmente porque os preços subiram, sem que haja crescimento real da produção.

No código abaixo, aplicamos a fórmula a cada preço da lista usando `map()` com uma função lambda — um caso típico em que lambda é mais conciso do que definir uma função completa com `def`:

```
precos_nominais    = [10.00, 12.50, 9.80, 15.00, 11.30]
inflacao_acumulada = 0.15 # 15% de inflação acumulada no período

precos_reais = list(map(lambda p: p / (1 + inflacao_acumulada), precos_nominais))

print("Preço nominal -> Preço real (deflacionado)")
for nominal, real in zip(precos_nominais, precos_reais):
    print(f"  R$ {nominal:.2f} -> R$ {real:.2f}")
```

```
Preço nominal -> Preço real (deflacionado)
  R$ 10.00 -> R$ 8.70
  R$ 12.50 -> R$ 10.87
  R$ 9.80 -> R$ 8.52
  R$ 15.00 -> R$ 13.04
  R$ 11.30 -> R$ 9.83
```

5.8.2 Função `filter()`

`filter()` seleciona os elementos de uma sequência que satisfazem um critério booleano:

```
numeros       = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numeros_pares = list(filter(lambda x: x % 2 == 0, numeros))
print(numeros_pares)
```

```
[2, 4, 6, 8, 10]
```

5.8.2.1 Exemplo: países com inflação acima da meta

```
paises_inflacao = [
    ("Brasil",    4.8),
    ("Chile",     3.2),
    ("Argentina", 211.0),
    ("México",    4.7),
    ("Colômbia",  9.3),
    ("Peru",      2.9),
```

```python
    ("Uruguai",   5.1),
]

meta_inflacao = 4.5

acima_da_meta = list(filter(lambda x: x[1] > meta_inflacao, paises_inflacao))

print(f"Países com inflação acima de {meta_inflacao}%:")
for pais, inf in acima_da_meta:
    print(f"  {pais}: {inf}%")
```

```
Países com inflação acima de 4.5%:
  Brasil: 4.8%
  Argentina: 211.0%
  México: 4.7%
  Colômbia: 9.3%
  Uruguai: 5.1%
```

5.8.3 Função `sorted()`

`sorted()` ordena sequências. O parâmetro `key` aceita uma função que define o critério de ordenamento.

> **O método `.items()`**
>
> Dicionários em Python armazenam pares `chave: valor`. O método `.items()` retorna esses pares como uma sequência de tuplas `(chave, valor)`, o que permite iterá-los e ordená-los como qualquer outra sequência.

```python
# Ordenando uma lista de tuplas pelo segundo elemento
minha_lista          = [("maçã", 1), ("banana", 3), ("laranja", 2)]
minha_lista_ordenada = sorted(minha_lista, key=lambda x: x[1])
print(minha_lista_ordenada)

# Ordenando um dicionário pelos valores (de maior para menor)
notas = {"Danilo": 7, "Mariana": 9, "Bruno": 4, "Daniela": 8.5}

notas_ordenadas = sorted(
    notas.items(),        # retorna [(chave, valor), ...]
    key=lambda x: x[1],  # ordena pelo valor (segundo elemento de cada tupla)
```

```
    reverse=True
)
print(notas_ordenadas)
```

```
[('maçã', 1), ('laranja', 2), ('banana', 3)]
[('Mariana', 9), ('Daniela', 8.5), ('Danilo', 7), ('Bruno', 4)]
```

5.8.3.1 Exemplo: ranking de países por PIB per capita

```
# (país, PIB per capita em US$ mil, taxa de desemprego em %)
paises = [
    ("Brasil",    9.0,  6.2),
    ("Argentina", 11.6, 7.7),
    ("Chile",     16.5, 8.9),
    ("México",    10.8, 2.7),
    ("Colômbia",   6.8, 9.1),
    ("Peru",       6.6, 7.4),
]

# Por PIB per capita (maior → menor)
ranking_pib = sorted(paises, key=lambda x: x[1], reverse=True)

print("Ranking por PIB per capita (US$ mil):")
for i, (pais, pib, _) in enumerate(ranking_pib, start=1):
    print(f"  {i}º {pais}: US$ {pib:.1f} mil")

print()

# Por taxa de desemprego (menor → maior)
ranking_desemp = sorted(paises, key=lambda x: x[2])

print("Ranking por taxa de desemprego (menor → maior):")
for i, (pais, _, desemp) in enumerate(ranking_desemp, start=1):
    print(f"  {i}º {pais}: {desemp}%")
```

```
Ranking por PIB per capita (US$ mil):
  1º Chile: US$ 16.5 mil
  2º Argentina: US$ 11.6 mil
  3º México: US$ 10.8 mil
```

```
  4º Brasil: US$ 9.0 mil
  5º Colômbia: US$ 6.8 mil
  6º Peru: US$ 6.6 mil

Ranking por taxa de desemprego (menor → maior):
  1º México: 2.7%
  2º Brasil: 6.2%
  3º Peru: 7.4%
  4º Argentina: 7.7%
  5º Chile: 8.9%
  6º Colômbia: 9.1%
```

> **ℹ Nota**
>
> **Sobre a função `enumerate()`**
> A função `enumerate()` permite percorrer uma lista obtendo ao mesmo tempo o índice e o valor de cada elemento. No exemplo, ela é usada para gerar automaticamente a posição no ranking (1º, 2º, etc.), começando em 1 com o parâmetro `start=1`, sem precisar controlar manualmente um contador.

5.9 Exercícios

1. Escreva uma função `variacao_percentual(valor_inicial, valor_final)` que calcule a variação percentual entre dois valores. Aplique-a para calcular a variação do IPCA entre dois anos hipotéticos e a variação do PIB real entre dois trimestres.

2. Usando as funções `calcular_inss` e `calcular_irpf` definidas neste capítulo, escreva uma função `aliquota_efetiva(salario_bruto)` que retorne a **alíquota efetiva total** (INSS + IRPF como percentual do salário bruto). Exiba a alíquota efetiva para os salários `[2_000, 5_000, 10_000, 20_000]` e comente o que você observa.

3. Escreva uma função `elasticidade_preco(q1, q2, p1, p2)` que calcule a **elasticidade-preço da demanda** pela fórmula:

$$\varepsilon = \frac{\Delta Q / Q_1}{\Delta P / P_1}$$

Teste a função com os seguintes dados: quando o preço de um bem passa de R$10 para R$12, a quantidade demandada cai de 100 para 80 unidades. Interprete o resultado.

4. Considere a lista de países e seus respectivos PIBs per capita (em US$ mil): `[("Brasil", 9.0), ("Chile", 16.5), ("Argentina", 11.6), ("México", 10.8), ("Peru", 6.6), ("Colômbia", 6.8)]`. Utilizando `filter()` e `map()` com funções lambda:

   a. Filtre apenas os países com PIB per capita acima de US$ 10 mil.
   b. Para os países filtrados, crie uma nova lista com o PIB per capita convertido para reais (use a taxa de câmbio de R$ 5,80/US$).

5. Expanda a função `simular_tcl()` definida neste capítulo para aceitar também o parâmetro `distribuicao`, que pode ser `"uniforme"` ou `"exponencial"`. Use `np.random.uniform(0, 1, n)` para a distribuição uniforme e `np.random.exponential(scale=1, size=n)` para a exponencial. Verifique se a média das médias converge para o valor teórico esperado em cada caso ($\mu = 0{,}5$ para a uniforme e $\mu = 1$ para a exponencial com `scale=1`).

6 Exceções

Todo programa, em algum momento, se depara com situações inesperadas: o usuário digita um texto onde se esperava um número, um arquivo não existe no caminho indicado, uma divisão por zero é tentada. Em vez de simplesmente travar, o Python possui um mecanismo estruturado para lidar com esses casos — as **exceções**.

Aprender a ler e tratar erros é uma das habilidades mais práticas da programação. Um bom código precisa ser **defensivo**: deve antecipar que as entradas podem estar mal formatadas e responder de forma controlada, sem interromper abruptamente a análise.

> Pense nas exceções como **cláusulas de contingência** em um contrato. Um contrato bem redigido não ignora a possibilidade de imprevistos — ele os prevê e especifica o que fazer em cada cenário. Da mesma forma, um bom programa não assume que tudo vai correr perfeitamente: ele define o que acontece quando algo dá errado.

6.1 Os erros mais comuns em Python

A tabela abaixo resume os principais tipos de exceção que você pode encontrar ao programar em Python, com uma breve explicação e exemplos didáticos:

Exceção	Quando ocorre	Exemplo
`SyntaxError`	Erro de sintaxe (regra de escrita)	`print("olá)`
`NameError`	Uso de nome/variável não definida	`print(x)` (sem definir `x` antes)
`TypeError`	Operação entre tipos incompatíveis	`"1500" + 8300`
`ValueError`	Valor inadequado para uma operação	`int("abc")`
`ZeroDivisionError`	Divisão por zero	`100 / 0`
`IndexError`	Índice inexistente em lista	`lista[99]` com lista de 3 itens
`KeyError`	Chave inexistente em dicionário	`dados["inflacao"]` sem essa chave
`IndentationError`	Indentação incorreta	Código fora de alinhamento
`AttributeError`	Acesso a atributo que não existe	`100.upper()`
`FileNotFoundError`	Arquivo não encontrado	`open("dados.csv")` sem o arquivo

Esses são apenas alguns exemplos. Ler e interpretar corretamente as mensagens de erro é fundamental para depurar e evoluir seu código com confiança.

6.2 Erros de sintaxe (`SyntaxError`)

O primeiro tipo de erro que você provavelmente irá encontrar ao programar em Python é o **erro de sintaxe**. Ele ocorre antes mesmo do programa começar a rodar: o Python tenta interpretar o código e encontra algo que não segue as regras da linguagem — como uma aspa faltando, um parêntese não fechado ou uma indentação incorreta.

O exemplo clássico:

```python
print("Olá, Mundo!)
```

```
SyntaxError: EOL while scanning string literal (377803247.py, line 1)
  Cell In[1], line 1
    print("Olá, Mundo!)
                       ^
SyntaxError: EOL while scanning string literal
```

O Python aponta exatamente a linha e o caractere onde detectou o problema. A mensagem `EOL while scanning string literal` significa que chegou ao fim da linha (*End Of Line*) sem encontrar o fechamento da string.

> Como ler uma mensagem de erro
>
> Toda mensagem de erro do Python segue a mesma estrutura:
>
> 1. **`File "...", line N`**: arquivo e linha onde o erro ocorreu.
> 2. **Trecho do código**: a linha problemática, com ^ apontando o local exato.
> 3. **Tipo do erro**: `SyntaxError`, `ValueError`, `NameError`.
> 4. **Descrição**: explicação em inglês do que deu errado.
>
> Sempre leia de baixo para cima: o tipo e a descrição são o ponto de partida para entender o problema.

`SyntaxError` não pode ser resolvido com `try/except`, pois o programa nem chega a ser executado. A única solução é revisar e corrigir o código.

6.3 Erros em tempo de execução

Ao contrário dos erros de sintaxe, os **erros de tempo de execução** (*runtime errors*) ocorrem enquanto o programa já está rodando. O código está sintaticamente correto, mas algo inesperado acontece durante a execução — e aí o Python **lança uma exceção**.

6.3.1 `ValueError`

Um `ValueError` ocorre quando uma função recebe um argumento do tipo certo, mas com um **valor inadequado**. O caso mais comum é tentar converter algo que não é número em inteiro ou decimal:

```
x = int(input("Digite um número: "))
print(f"O número digitado foi {x}")
```

Como programadores, devemos ser cautelosos para garantir que os usuários estejam inserindo exatamente o que esperamos. Se executarmos esse programa e digitarmos “fea”, veremos: `ValueError: invalid literal for int() with base 10: 'fea'`.

Em outras palavras, a função int não consegue converter o texto “fea” em um número. Uma estratégia eficaz para corrigir esse possível erro é criar um tratamento de erros para garantir que o usuário se comporte conforme esperamos.

6.4 `try` e `except`

Para lidar com exceções em tempo de execução, o Python oferece a estrutura `try/except`. A ideia é simples: colocamos no bloco `try` o código que pode falhar e no bloco `except` o que deve acontecer *se* a exceção ocorrer:

```
try:
    <código que pode gerar exceção>
except <TipoDeExcecao>:
    <o que fazer se a exceção ocorrer>
```

Por exemplo:

```
try:
    x = int(input("Digite um número: "))
    print(f"O número digitado foi {x}")
except ValueError:
    print("Valor inválido. Por favor, digite um número inteiro.")
```

Se o usuário digitar `"dez"` em vez de `10`, o `int()` lançaria um `ValueError`. Com `try/except`, capturamos esse erro e exibimos uma mensagem amigável em vez de travar o programa.

> ⚠ Minimize o bloco `try`
>
> Uma boa prática é colocar dentro do `try` apenas **a linha que pode falhar**, não o bloco inteiro. Isso evita capturar exceções de linhas que não têm nada a ver com o erro esperado.

6.4.1 O problema do `NameError` com `try/except`

Note que essa ainda não é a melhor forma de implementar esse código, pois estamos tentando executar duas linhas de código. Veja o que acontece se colocarmos o `print` fora do `try`:

```
try:
    x = int(input("Digite um número: "))
except ValueError:
    print("Valor inválido. Por favor, digite um número inteiro.")

print(f"O número digitado foi {x}")
```

Observe que, embora essa versão siga a boa prática de colocar no `try` apenas a linha que pode falhar, ela introduz um novo problema. Se o usuário digitar algo inválido, ocorre um `ValueError`, que é tratado pelo `except`. No entanto, logo depois disso, o programa tenta executar `print(f"O número digitado foi {x}")`.

Nesse caso, aparece um `NameError`, pois a variável `x` **não chegou a ser criada**. Isso acontece porque, em `x = int(input("Digite um número: "))`, o Python primeiro avalia o lado direito da atribuição — isto é, tenta executar `int(input(...))`. Somente se essa etapa der certo o valor será atribuído a `x`.

Se a conversão para inteiro falhar, a atribuição nunca acontece. Portanto, ao chegar no `print`, o nome `x` ainda não existe na memória. É exatamente para resolver esse tipo de situação que o bloco `else` é útil: ele garante que o código que depende do sucesso do `try` só será executado quando nenhuma exceção ocorrer.

6.5 `else`

O bloco `else` em uma estrutura `try/except` executa **somente se nenhuma exceção ocorreu**. Ele é o lugar certo para o código que depende do sucesso do `try`:

```python
try:
    x = int(input("Digite um número: "))
except ValueError:
    print("Valor inválido. Por favor, digite um número inteiro.")
else:
    print(f"O número digitado foi {x}")
```

Observe que, **se nenhuma exceção ocorrer**, o programa executará o bloco `else`. Ao rodar o código e digitar `10`, por exemplo, a conversão será bem-sucedida e o valor será exibido normalmente. Já ao digitar `"dez"`, a exceção será capturada pelo `except`, evitando que o programa quebre de forma inesperada.

Embora essa versão já seja mais segura, ainda há espaço para melhorar a experiência do usuário. No estado atual, se a entrada for inválida, o programa apenas exibe a mensagem de erro e termina. Em muitos casos, o comportamento ideal é **dar ao usuário uma nova chance**, repetindo a solicitação até que ele forneça um número inteiro válido.

Para isso, podemos combinar o tratamento de exceções com um **loop**, criando um mecanismo de validação que continua pedindo a entrada correta até que a conversão seja feita com sucesso.

6.5.1 Loop de validação com `else`

Um padrão muito comum é combinar `while True` com `try`/`except`/`else` para continuar pedindo entrada até o usuário fornecer um valor válido:

```python
while True:
    try:
        x = int(input("Digite um número: "))
    except ValueError:
        print("Valor inválido. Por favor, digite um número inteiro.")
    else:
        break

print(f"O número digitado foi {x}")
```

O comando `while True` cria um laço que continuará pedindo a entrada do usuário indefinidamente, até que uma entrada válida seja fornecida.

Quando o usuário digita um valor correto, o `break` interrompe o loop e o programa segue para a próxima etapa, exibindo o resultado. Dessa forma, se o usuário errar, ele recebe uma mensagem de orientação e tem novas chances de acertar, tornando a experiência mais amigável e robusta. Podemos melhorar o código da seguinte maneira:

```
while True:
    try:
        x = int(input("Digite um número: "))
        break
    except ValueError:
        print("Valor inválido. Por favor, digite um número inteiro.")

print(f"O número digitado foi {x}")
```

6.5.2 Utilizando funções

A combinação de `try`, `except`, `else` e `break` se torna especialmente poderosa quando encapsulada em funções reutilizáveis.

```
def main():
    x = obter_inteiro()
    print(f"O número digitado foi {x}")

def obter_inteiro():
    while True:
        try:
            x = int(input("Digite um número: "))
        except ValueError:
            print("Valor inválido. Por favor, digite um número inteiro.")
        else:
            break
    return x

main()
```

Observe que, ao criar uma função para obter um número inteiro, tornamos o programa mais organizado e reutilizável. Agora, a lógica principal do programa se resume a poucas linhas, facilitando a leitura e a manutenção.

Ainda assim, sempre é possível aprimorar o código. Que outras melhorias poderiam ser feitas para tornar essa função ainda mais robusta, flexível ou amigável para o usuário?

```
def main():
    x = obter_inteiro()
    print(f"O número digitado foi {x}")

```

```
def obter_inteiro():
    while True:
        try:
            x = int(input("Digite um número: "))
        except ValueError:
            print("Valor inválido. Por favor, digite um número inteiro.")
        else:
            return x


main()
```

Note que o comando **`return`** não apenas encerra o loop, mas também devolve o valor desejado para quem chamou a função. Algumas pessoas podem argumentar que seria possível obter o mesmo resultado de outras formas, como por exemplo:

```
def main():
    x = obter_inteiro()
    print(f"O número digitado foi {x}")


def obter_inteiro():
    while True:
        try:
            return int(input("Digite um número: "))
        except ValueError:
            print("Valor inválido. Por favor, digite um número inteiro.")


main()
```

6.6 pass

Às vezes, queremos capturar uma exceção mas **não fazer nada** — simplesmente continuar o programa sem exibir mensagem de erro. Para isso usamos `pass`:

```
try:
    <código que pode falhar>
except <Exceção>:
    pass   # ignora o erro silenciosamente
```

O `pass` é útil quando o comportamento correto diante de um erro é simplesmente tentar de novo (em um loop) ou pular aquele item.

> **`pass` vs. mensagem de erro**
>
> Use `pass` quando o erro é esperado e ignorá-lo é o comportamento correto (como pular entradas inválidas em uma lista). Prefira `except ... : print(...)` quando o erro é inesperado e o usuário precisa ser informado. Nunca use `pass` apenas para "silenciar" um erro que você não entendeu — isso esconde problemas reais.

```
def main():
    x = obter_inteiro()
    print(f"O número digitado foi {x}")


def obter_inteiro():
    while True:
        try:
            return int(input("Digite um número: "))
        except ValueError:
            pass


main()
```

Note que o código ainda funciona, mas não informa repetidamente ao usuário qual foi o erro cometido. Em algumas situações, é importante ser claro e mostrar ao usuário qual erro ocorreu; em outras, pode ser suficiente apenas pedir a entrada novamente, sem exibir mensagens.

Uma última melhoria interessante para a função `obter_inteiro` é permitir que o texto do prompt seja personalizado. Assim, não dependemos de um texto fixo dentro da função e tornamos o código mais flexível e reutilizável. Modifique sua implementação para aceitar um parâmetro de prompt.

```
def main():
    x = obter_inteiro("Digite um número: ")
    print(f"O número digitado foi {x}")


def obter_inteiro(prompt):
    while True:
        try:
```

```
            return int(input(prompt))
        except ValueError:
            pass


main()
```

Você pode aprender mais na documentação oficial do Python sobre o comando pass.

6.6.1 Exemplo Prático: Limpeza de base de dados

Suponha que você recebeu um CSV com dados mensais de inflação (IPCA). A coluna "ipca_bruto" deveria ser numérica, mas veio com problemas:

- valores vazios
- textos ("n/d", "-")
- números com vírgula ou espaço

```python
# Série do IPCA com valores mal formatados (dados "sujos")
ipca_bruto = ["4,62", "3,15", ".", "5,79", "-", "4,31", "", "3.72", "n/d", "6.50", "  4.83 "]

ipca_limpo = []

for valor in ipca_bruto:
    try:
        ipca_limpo.append(float(valor.strip().replace(",", ".")))
    except ValueError:
        pass # ignora valores não numéricas sem interromper o loop

print(f"Valores originais:    {len(ipca_bruto)}")
print(f"Valores válidos:      {len(ipca_limpo)}")
print(f"Valores ignorados:    {len(ipca_bruto) - len(ipca_limpo)}")
print(f"IPCA médio (válidos): {sum(ipca_limpo) / len(ipca_limpo):.2f}%")
```

```
Valores originais:    11
Valores válidos:      7
Valores ignorados:    4
IPCA médio (válidos): 4.70%
```

6.7 raise

Em algumas situações, você pode querer interromper a execução do seu programa de forma controlada, lançando uma exceção quando uma condição indesejada ocorre. Para isso, usamos a palavra-chave `raise`.

No exemplo a seguir, a função `calcular_raiz_quadrada()` lança uma exceção `ValueError` se o número fornecido for negativo, garantindo que apenas valores válidos sejam processados:

```
def calcular_raiz_quadrada(numero):
    if numero < 0:
        raise ValueError("O número deve ser positivo")
    return numero ** 0.5

valor = float(input("Digite um número: "))
resultado = calcular_raiz_quadrada(valor)
print(f"A raiz quadrada de {valor} é {resultado}")
```

Além de utilizar as exceções já existentes no Python, também podemos criar nossas próprias exceções personalizadas para lidar com situações específicas do nosso programa. Para isso, basta criar uma nova classe que herda de `Exception`. Assim, conseguimos definir mensagens e comportamentos sob medida para os nossos casos de uso, tornando o tratamento de erros ainda mais claro e organizado:

```
def verificar_valor(valor):
    if valor < 0:
        raise Exception("O número não pode ser negativo!")

verificar_valor(-5)
```

6.8 Capturando múltiplas exceções

Um mesmo bloco `try` pode capturar diferentes tipos de exceção, cada um com seu próprio tratamento:

```
try:
    <código>
except TipoA:
    <tratamento para TipoA>
except TipoB:
    <tratamento para TipoB>
```

No exemplo a seguir, o bloco `try` tenta executar a divisão de dois números fornecidos pelo usuário. Se o usuário tentar dividir por zero, o except `ZeroDivisionError` captura essa exceção e exibe uma mensagem de erro. Além disso, se o usuário digitar algo que não seja um número, o except `ValueError` captura essa exceção e exibe uma mensagem de erro apropriada.

```
try:
    x = int(input("Digite um número: "))
    y = int(input("Digite outro número: "))
    resultado = x / y
    print(f"O resultado é: {resultado}")
except ZeroDivisionError:
    print("Erro: Não é possível dividir por zero!")
except ValueError:
    print("Erro: Entrada inválida! Por favor, digite um número.")
```

Note que podemos capturar vários tipos ao mesmo tempo com uma tupla:

```
except (TipoA, TipoB):
    <tratamento comum para ambos>
```

```
try:
    x = int(input("Digite um número: "))
    y = int(input("Digite outro número: "))
    resultado = x / y
    print(f"O resultado é: {resultado}")
except (ZeroDivisionError, ValueError):
    print("Erro: Certifique-se de digitar apenas números e que o divisor não seja zero.")
```

6.9 Exercícios

1. O código abaixo contém três erros — um `SyntaxError`, um `NameError` e um `ValueError`. Identifique e corrija cada um deles:

```
def calcular_variacao(v_inicial, v_final)
    variacao = (v_final - v_inicial) / v_inicial * 100
    return variacao

pib_2022 = float("8.900,00")
pib_2023 = 9250.0

resultado = calcular_variacao(pib_2022, PIB_2023)
print(f"Variação do PIB: {resultado:.2f}%")
```

2. Escreva uma função `ler_taxa(prompt)` que solicite ao usuário uma taxa de juros entre 0% e 100%. A função deve:

    - Usar `try/except` para capturar `ValueError` se a entrada não for numérica.
    - Usar `else` para verificar se o valor está no intervalo permitido, pedindo novamente caso contrário.
    - Retornar o valor em decimal (ex: entrada `10.5` → retorna `0.105`).

3. Dada a lista abaixo com dados de exportações brasileiras (em US$ bilhões), alguns valores estão corrompidos. Escreva um programa que, usando `try/except/pass`:

    - Leia os valores válidos.
    - Calcule e exiba o total e a média das exportações válidas.
    - Informe quantos registros foram ignorados.

    ```
    exportacoes = ["32.1", "28.7", "N/A", "35.4", "erro", "29.8",
                   "31.2", "--",   "33.6", "27.9", "34.1", "30.5"]
    ```

4. Uma planilha de preços de commodities contém o seguinte dicionário. Escreva um programa que tente acessar cada chave de uma lista de consultas, capturando `KeyError` para as chaves inexistentes e `ValueError` para valores não numéricos, exibindo uma mensagem específica para cada caso.

    ```
    commodities = {
        "soja":     "117.50",
        "milho":    "48.20",
        "cafe":     "pendente",
        "petroleo": "88.75",
    }
    consultas = ["soja", "milho", "cafe", "algodao", "petroleo", "trigo"]
    ```

5. Expanda a função `processar_serie()` definida nesta aula para aceitar um parâmetro opcional `limiar_invalidos` (padrão: `0.3`). Se a proporção de dados inválidos exceder esse limiar, a função deve lançar um `ValueError` com uma mensagem descritiva em vez de retornar o resultado parcial. Teste com a série `["1.2", "n/d", "n/d", "n/d", "0.8"]`.

7 Expressões Regulares

Expressões regulares (ou *regular expressions*, frequentemente abreviadas como regex) são sequências de caracteres usadas para verificar se um padrão existe em um texto (string).

Elas são amplamente usadas, por exemplo, para validar formatos, extrair informações específicas ou transformar dados.

Neste material, você vai aprender os principais conceitos de expressões regulares em Python. Começaremos importando o módulo `re`, responsável pelo suporte a regex na linguagem. Em seguida, você verá:

- como usar caracteres simples para fazer correspondências;
- como funcionam os metacaracteres (caracteres especiais);
- como aplicar repetições nos padrões;
- como criar grupos de captura e grupos nomeados;
- como utilizar as principais funções do módulo `re`, como `search`, `match`, `findall`, `finditer`, `split` e `sub`;
- como usar flags (modificadores) para ajustar o comportamento das buscas.

Ao longo do capítulo, exemplos práticos mostram como as expressões regulares podem ser aplicadas para resolver problemas comuns de extração, validação e limpeza de dados.

7.1 O módulo `re`

Em Python, as expressões regulares são suportadas pelo módulo `re`. Isso significa que, para começar a utilizá-las em seus scripts, é necessário importar esse módulo usando `import`:

```
import re
```

A biblioteca `re` em Python oferece diversas funções que fazem dela uma habilidade que vale a pena dominar. Todas recebem como primeiro argumento um **padrão**, ou seja, uma string especial que descreve o que procurar.

Função	O que faz
`re.search(padrão, string)`	Procura o padrão em qualquer posição da string; retorna um objeto `Match` ou `None`
`re.match(padrão, string)`	VVerifica se o padrão ocorre no início da string
`re.findall(padrão, string)`	Retorna todas as ocorrências do padrão como uma lista
`re.finditer(padrão, string)`	Retorna um iterador de objetos `Match`, permitindo acessar detalhes como posição e grupos de cada ocorrência
`re.split(padrão, string)`	Divide a string nos pontos onde o padrão é encontrado
`re.sub(padrão, substituição, string)`	Substitui todas as ocorrências do padrão por outro texto

Veremos a seguir a linguagem desses padrões. Neste material, você verá algumas delas em detalhes.

7.2 Sintaxe dos padrões

A "linguagem" das expressões regulares é formada por **metacaracteres** — símbolos com significado especial.

7.2.1 Metacaracteres de correspondência

Símbolo	Significado
`.`	Qualquer caractere, exceto quebra de linha
`\d`	Dígito decimal (`[0-9]`)
`\D`	Qualquer caractere que não é dígito
`\w`	Caractere alfanumérico ou `_` (`[a-zA-Z0-9_]`)
`\W`	Qualquer caractere que não é alfanumérico
`\s`	Espaço em branco (espaço, tab, `\n`)
`\S`	Qualquer caractere que não é espaço em branco
`[abc]`	Qualquer um dos caracteres listados
`[^abc]`	Qualquer caractere exceto os listados
`[a-z]`	Qualquer caractere dentro do intervalo
`\b`	Limite de palavra (início ou fim de palavra)
`\t`	Tabulação

Símbolo	Significado
\n	Nova linha
\r	Corresponde ao return

7.2.2 Metacaracteres de quantidade

Símbolo	Significado
*	0 ou mais repetições
+	1 ou mais repetições
?	0 ou 1 repetição (torna o elemento opcional)
{m}	Exatamente m repetições
{m,n}	Entre m e n repetições

7.2.3 Âncoras, grupos e controle

Símbolo	Significado
^	Início da string
$	Fim da string
\A	Início absoluto da string
\Z	Fim absoluto da string
A|B	A ou B
(...)	Grupo de captura
(?:...)	Grupo sem captura
(?P<nome> ...)	Grupo nomeado
\	Escape: remove o significado especial do próximo caractere

Strings brutas (r"...")

Em Python, o caractere \ tem significado especial em strings comuns (por exemplo, \n representa nova linha e \t representa tabulação). Para evitar conflitos com os metacaracteres das expressões regulares (como \d, \w, \s), é recomendável escrever padrões regex usando strings brutas, com o prefixo r"...". Assim, o Python não interpreta essas sequências como escapes da linguagem:

```
re.search(r"\d+", texto)    # correto - \d é interpretado pelo re
re.search("\d+", texto)     # ambíguo - Python tenta interpretar \d antes
```

7.3 `re.search`

A função `re.search` percorre toda a string em busca da primeira posição onde o padrão corresponde. Se encontrar uma correspondência, retorna um objeto `Match`; caso contrário, retorna `None`.

Diferente de `re.findall`, que retorna todas as ocorrências, `re.search` interrompe a busca ao encontrar a primeira correspondência — sendo mais eficiente quando o objetivo é apenas verificar se o padrão existe na string.

```
import re

texto = "O IPCA de março foi de 0,83%"

resultado = re.search("[0-9]+,[0-9]+", texto)
resultado = re.search(r"\d+,\d+", texto)
print(resultado)           # objeto Match
print(resultado.group())   # trecho encontrado
```

```
<re.Match object; span=(23, 27), match='0,83'>
0,83
```

7.4 Agrupamento em Expressões Regulares

O recurso de **agrupamento** em expressões regulares permite capturar partes específicas do texto que corresponde ao padrão.

Trechos do padrão delimitados por parênteses `()` são chamados de **grupos de captura**. Esses parênteses não alteram o que será correspondido, mas organizam o resultado em partes que podem ser acessadas separadamente. É possível usar o método `.group()` para acessar essas partes:

- `.group()` (ou `.group(0)`): retorna o trecho completo correspondente ao padrão
- `.group(n)`: retorna um grupo específico (o n-ésimo grupo)
- `.groups()`: retorna todos os grupos como tupla

Considere um exemplo simples: imagine que precisamos extrair o dia, o mês e o ano de um trecho de texto. Nesse caso, faz sentido usarmos grupos de captura para isolar cada uma dessas partes, permitindo acessá-las individualmente a partir do resultado da correspondência.

```python
texto = "A reunião do Copom em 18/03/2025 definiu a nova taxa básica de juros."

resultado = re.search(r"(\d{2})/(\d{2})/(\d{4})", texto)

if resultado:
    print(f"Data: {resultado.group()}")
    print(f"Dia: {resultado.group(1)}")
    print(f"Mês: {resultado.group(2)}")
    print(f"Ano: {resultado.group(3)}")
    print(f"Grupos: {resultado.groups()}")
```

```
Data: 18/03/2025
Dia: 18
Mês: 03
Ano: 2025
Grupos: ('18', '03', '2025')
```

Outra forma de fazer isso é utilizando a sintaxe com `(?P<nome>...)`, que permite criar **grupos nomeados**.

Grupos nomeados tornam o código mais legível e fácil de entender, pois você pode acessar cada parte pelo nome, em vez de depender da posição numérica.

Vamos ver isso na prática usando o mesmo exemplo anterior.

```python
texto = "A reunião do Copom em 18/03/2025 definiu a nova taxa básica de juros."

resultado = re.search(r"(?P<dia>\d{2})/(?P<mes>\d{2})/(?P<ano>\d{4})", texto)

if resultado:
    print(f"Data: {resultado.group()}")
    print(f"Dia: {resultado.group(1)}")
    print(f"Mês: {resultado.group(2)}")
    print(f"Ano: {resultado.group(3)}")
    print(f"Grupos: {resultado.groups()}")
```

```
Data: 18/03/2025
Dia: 18
Mês: 03
Ano: 2025
Grupos: ('18', '03', '2025')
```

Uma forma simples e elegante de refinar o código anterior é usar o **operador morsa** (`:=`), que permite atribuir e testar um valor ao mesmo tempo:

> **O operador morsa (`:=`)**
>
> Permite atribuir o resultado de uma expressão a uma variável dentro de uma condição. É especialmente útil com `re.search`, pois evita repetir a chamada e o nome da variável:
>
> ```
> if resultado := re.search(r"...", texto):  # atribui e testa em uma linha
>     print(resultado.group())
> ```

Cem o operador morsa (`:=`), o código anterior ficaria assim:

```
texto = "A reunião do Copom em 18/03/2025 definiu a nova taxa básica de juros."

if resultado := re.search(r"(?P<dia>\d{2})/(?P<mes>\d{2})/(?P<ano>\d{4})", texto):
    print(f"Data: {resultado.group()}")
    print(f"Dia: {resultado.group(1)}")
    print(f"Mês: {resultado.group(2)}")
    print(f"Ano: {resultado.group(3)}")
    print(f"Grupos: {resultado.groups()}")

# Também é possível recuperar os grupos nomeados como um dicionário usando groupdict():
resultado.groupdict()
```

```
Data: 18/03/2025
Dia: 18
Mês: 03
Ano: 2025
Grupos: ('18', '03', '2025')
```

```
{'dia': '18', 'mes': '03', 'ano': '2025'}
```

7.5 `re.match`

A função `re.match` é semelhante a `re.search`, mas com uma diferença importante: ela só verifica se o padrão aparece no **início da string**. Na prática, isso equivale a usar ^ no começo do padrão com `re.search`.

```
codigo_ibge = "Código IBGE: 3550308" # código do município de São Paulo

# search procura em qualquer posição da string
print("Usando search:", re.search(r"\d{7}", codigo_ibge).group())

# match tenta encontrar o padrão apenas no início
print("Usando match:", re.match(r"\d{7}", codigo_ibge))
print("Usando match:", re.match(r"Código", codigo_ibge))
print("Usando match:", re.match(r"Código", codigo_ibge).group())
```

```
Usando search: 3550308
Usando match: None
Usando match: <re.Match object; span=(0, 6), match='Código'>
Usando match: Código
```

7.6 `re.findall`

A função `re.findall` percorre toda a string em busca de correspondências e retorna todas as ocorrências encontradas como uma lista. Cada elemento dessa lista representa uma correspondência do padrão.

Como o `re.findall` não retorna objetos `Match`, mas sim os resultados já extraídos (strings ou tuplas), não é possível usar os métodos `group()` e `groups()`. Isso acontece porque cada elemento retornado já contém diretamente os dados capturados — ou seja, os grupos já vêm "prontos" no resultado, tornando desnecessário o uso de `.group()` ou `.groups()`.

O formato dos elementos retornados depende do uso de grupos de captura `()` no padrão:

- **Sem grupos de captura**: retorna uma lista de strings com os trechos completos de correspondência
- **Com 1 grupo de captura**: retorna uma lista de strings contendo apenas o conteúdo desse grupo
- **Com múltiplos grupos**: retorna uma lista de tuplas, onde cada tupla corresponde a uma ocorrência e contém os grupos capturados

Dica:

Use grupos de captura quando quiser extrair partes específicas do texto. Isso torna o resultado do `re.findall` mais flexível e especialmente útil para análise e processamento de dados.

```python
texto = """
Segundo dados divulgados nesta quinta-feira, o IPCA registrou alta de 4,83% em 2025, enquanto
"""

# 1) Sem grupos: retorna as correspondências completas
print("Sem grupos:")
print(re.findall(r"\d+,\d+%", texto))
print()

# Com 1 grupo: retorna apenas o conteúdo do grupo
print("Com 1 grupo:")
print(re.findall(r"(\d+,\d+%)", texto))
print()

# Com múltiplos grupos: retorna lista de tuplas (índice, valor)
print("Com múltiplos grupos (índice + valor):")
padrao = r"\b(IPCA|IGP-M|INPC)\b.{0,50}?(\d{1,3},\d+)%"

resultados = re.findall(padrao, texto)
print(resultados)
print()

for indice, valor in resultados:
    print(f"{indice}: {valor}%")
```

```
Sem grupos:
['4,83%', '7,12%', '4,61%']

Com 1 grupo:
['4,83%', '7,12%', '4,61%']

Com múltiplos grupos (índice + valor):
[('IPCA', '4,83'), ('IGP-M', '7,12'), ('INPC', '4,61')]

IPCA: 4,83%
IGP-M: 7,12%
INPC: 4,61%
```

7.7 re.finditer

A função `re.finditer` é semelhante ao `re.findall`: ela encontra **todas as ocorrências** do padrão ao longo da string.

A diferença é que, em vez de retornar uma lista de resultados, ela retorna um **iterador de objetos** `Match`.

Em outras palavras, `re.finditer` permite percorrer todas as ocorrências e acessar detalhes de cada uma, como:

- o texto correspondente
- os grupos de captura
- a posição inicial e final na string

Dica:

`finditer()` é uma excelente escolha quando você precisa de mais informações sobre cada correspondência. Os objetos `Match` retornados contêm não apenas o texto encontrado, mas também sua posição na string original.

Além dos métodos `.group()`, `.groups()`, os objetos `Match` também possuem outros métodos úteis, como `.start()`, `.end()` e `.span()`.

```
texto = """
Segundo dados divulgados nesta quinta-feira, o IPCA registrou alta de 4,83% em 2025, enquanto
"""

padrao = r"\b(IPCA|IGP-M|INPC)\b.{0,50}?(\d{1,3},\d+)%"

for match in re.finditer(padrao, texto):
    print(match)
    print(match.group())
    print(f"Grupos: {match.groups()}")
    print(f"Índice: {match.group(1)}")
    print(f"Valor:  {match.group(2)}%")
    print(f"Posição: {match.span()}")
    print("-" * 20)
```

```
<re.Match object; span=(48, 76), match='IPCA registrou alta de 4,83%'>
IPCA registrou alta de 4,83%
Grupos: ('IPCA', '4,83')
Índice: IPCA
```

```
Valor:  4,83%
Posição: (48, 76)
--------------------
<re.Match object; span=(97, 117), match='IGP-M ficou em 7,12%'>
IGP-M ficou em 7,12%
Grupos: ('IGP-M', '7,12')
Índice: IGP-M
Valor:  7,12%
Posição: (97, 117)
--------------------
<re.Match object; span=(124, 151), match='INPC teve variação de 4,61%'>
INPC teve variação de 4,61%
Grupos: ('INPC', '4,61')
Índice: INPC
Valor:  4,61%
Posição: (124, 151)
--------------------
```

7.8 re.split

A função **re.split** divide a string sempre que o padrão é encontrado e retorna o resultado como uma lista. Se o argumento opcional **maxsplit** for diferente de zero, será realizado no máximo esse número de divisões.

Ela é mais poderosa que o .split() nativo do Python, pois aceita padrões complexos, como múltiplos delimitadores, espaços variáveis e combinações de caracteres.

Dica:

Use **re.split** para tratar textos “sujos” ou inconsistentes, em que os separadores não seguem um padrão fixo.

```
# .split() nativo: delimitador fixo
texto = "São Paulo/SP: 05508-010"
print("Usando .split():", texto.split(":"))
print([p.strip() for p in texto.split(":")])

# re.split: delimitador flexível - ponto e vírgula OU vírgula OU barra
texto = "Av. Professor Luciano Gualberto, 908; Butantã, São Paulo/SP: 05508-010"
print("Usando re.split():", re.split(r"[;,/]", texto))
```

```
print("Usando re.split():", re.split(r"\s*[;,/]\s*", texto))
print([p.strip() for p in re.split(r"[;,/]", texto)])
```

```
Usando .split(): ['São Paulo/SP', ' 05508-010']
['São Paulo/SP', '05508-010']
Usando re.split(): ['Av. Professor Luciano Gualberto', ' 908', ' Butantã', ' São Paulo', 'SP
010']
Usando re.split(): ['Av. Professor Luciano Gualberto', '908', 'Butantã', 'São Paulo', 'SP: 05
010']
['Av. Professor Luciano Gualberto', '908', 'Butantã', 'São Paulo', 'SP: 05508-
010']
```

7.9 `re.sub`

A função `re.sub` realiza substituições em uma string. Ela retorna uma nova string onde todas as ocorrências não sobrepostas do padrão são substituídas pelo valor definido em repl.

Se o padrão não for encontrado, a string original é retornada sem alterações.

Dica:

`re.sub` é indispensável para limpeza e normalização de dados, como remover caracteres indesejados, padronizar formatos ou ajustar textos antes de análises.

```
# Removendo formatação de valores monetários brasileiros
dados = ["R$ 1.234,56", "R$9.800,00", "$ 12.345.678,90"]

dados_limpos = []

for d in dados:
    sem_simbolo = re.sub(r"[R$\s]", "", d)       # remove R$ e espaços
    sem_milhar  = re.sub(r"\.", "", sem_simbolo) # remove ponto de milhar
    decimal   = re.sub(r",", ".", sem_milhar)     # substitui vírgula por ponto
    dados_limpos.append(float(decimal))

print(dados)
print(dados_limpos)
```

```
['R$ 1.234,56', 'R$9.800,00', '$ 12.345.678,90']
[1234.56, 9800.0, 12345678.9]
```

7.10 Flags

As **flags** permitem modificar o comportamento das buscas com expressões regulares, tornando-as mais flexíveis e adaptáveis a diferentes situações. Elas são passadas como um terceiro argumento nas funções do módulo `re`, por exemplo: `re.search(padrão, string, flags=re.I)`.

As principais flags são:

Flag	Efeito
`re.IGNORECASE`	Ignora a diferença entre maiúsculas e minúsculas (busca "case-insensitive")
`re.MULTILINE`	Faz com que `^` e `$` correspondam ao início e fim de cada linha, não só da string inteira
`re.DOTALL`	Faz com que o ponto `.` corresponda também à quebra de linha (`\n`)

Dica:

Você pode combinar várias flags usando o operador `|`, por exemplo: `flags=re.IGNORECASE | re.MULTILINE`.

```
texto = "A decisão do Copom de iniciar o ciclo de flexibilização monetária com um corte de 0

print(re.findall(r"Selic", texto))
print(re.findall(r"Selic", texto, re.IGNORECASE))
```

```
['Selic']
['selic', 'Selic']
```

```
texto = """
Aluno: Ana; Curso: Economia; Ingresso: 2023\naluno: Bruno; Curso: Administração; Ingresso: 20
"""

# Sem MULTILINE: ^ e $ correspondem ao início e fim de toda a string
print("MULTILINE")
print(re.findall(r"\d{4}$", texto))
print(re.findall(r"\d{4}$", texto, re.MULTILINE))
print()

# Sem DOTALL: . não casa com quebras de linha
print("DOTALL")
```

```python
print(re.search(r".+", texto).group())
print(re.search(r".+", texto, re.DOTALL).group())

# Com IGNORECASE e MULTILINE
print("IGNORECASE + MULTILINE")
print(re.findall(r"^aluno.+", texto))
print(re.findall(r"^aluno.+", texto, re.IGNORECASE))
print(re.findall(r"^aluno.+", texto, re.IGNORECASE | re.MULTILINE))
```

```
MULTILINE
['2022']
['2023', '2022']

DOTALL
Aluno: Ana; Curso: Economia; Ingresso: 2023

Aluno: Ana; Curso: Economia; Ingresso: 2023
aluno: Bruno; Curso: Administração; Ingresso: 2022

IGNORECASE + MULTILINE
[]
[]
['Aluno: Ana; Curso: Economia; Ingresso: 2023', 'aluno: Bruno; Curso: Administração; Ingress
```

7.11 Exercícios

1. Escreva uma função `validar_ticker(ticker)` que valide o formato de um ticker de ação brasileira na B3. Tickers válidos têm entre 4 e 6 letras maiúsculas seguidas de 1 ou 2 dígitos (ex: `"PETR4"`, `"VALE3"`, `"BBDC4"`, `"ABEV3"`). A função deve retornar `True` ou `False`.

2. Dado o texto abaixo, use `re.findall` para extrair **todas** as taxas percentuais mencionadas e calcule a média:

   ```python
   texto = """
   A rentabilidade do CDB foi de 12,5% ao ano, enquanto o Tesouro Selic
   rendeu 10,50%. Os fundos de renda fixa tiveram retorno médio de 11,2%,
   e as debêntures incentivadas pagaram 13,8% ao ano. A inflação (IPCA)
   ficou em 4,83%, resultando em um ganho real de aproximadamente 7,3%.
   """
   ```

3. A tabela abaixo contém dados de exportações com delimitadores inconsistentes. Use `re.split` para separar corretamente cada linha e calcule o total exportado:

```
dados = [
    "soja;32.450,00|EUA;jan/2024",
    "milho,12.800,50;China,fev/2024",
    "petroleo\t8.920,75|Argentina\tfev/2024",
]
```

4. Escreva uma função `extrair_cnpj(texto)` que use `re.findall` para encontrar **todos** os CNPJs em um texto no formato `XX.XXX.XXX/XXXX-XX`. Teste com um parágrafo que contenha múltiplos CNPJs misturados com texto.

5. Usando `re.sub`, escreva uma função `normalizar_serie(valores)` que receba uma lista de strings representando valores de uma série temporal — potencialmente com formatações inconsistentes como `"1.234,56"`, `"1234.56"`, `"R$ 1.234,56"` e `"1,234.56"` (formato americano) — e retorne uma lista de `float` com todos os valores normalizados.

References