

UNIP

UNIVERSIDADE PAULISTA

Algoritmos e Estrutura de Dados em Python

Autor: Prof. Tarcísio de Souza Peres

Colaboradores: Prof. Angel Antonio Gonzalez Martinez
Profa. Larissa Rodrigues Damiani

Professor conteudista: Tarcísio de Souza Peres

Formado em Engenharia da Computação e mestre pela Unicamp. Atualmente, é doutorando pelo Departamento de Letras Clássicas e Vernáculas da Faculdade de Filosofia, Ciências e Letras da USP. Atuou como pesquisador no Human Cancer Genome Project. Tem MBA em Gestão de TI pela Fiap e especialização em Gestão de Projetos pela FIA/USP. Cursou Gestão Estratégica e Competitividade pela Fundação Dom Cabral. Com perfil multidisciplinar, possui experiência nas áreas de tecnologia, saúde, governança, novos negócios e inovação. Trabalhou em multinacionais e órgãos públicos. É certificado pelo PMI, Cobit e autor de livro sobre mercado financeiro, além de livros de programação orientada a objetos básica e programação avançada em C#, desenvolvimento de software para a internet com ASP.Net, gestão de projetos ágeis, engenharia de requisitos e pensamento lógico computacional com Python. Atua como empreendedor (startups de inovação em tecnologia). É professor na UNIP, na cFGV e no CPS.

Dados Internacionais de Catalogação na Publicação (CIP)

P437a Peres, Tarcísio de Souza.

Algoritmos e Estrutura de Dados em Python / Tarcísio de Souza Peres. – São Paulo: Editora Sol, 2025.

260 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. Algoritmos. 2. Python. 3. Grafos. I. Título.

CDU 519.67

U522.51 – 25

Prof. João Carlos Di Genio
Fundador

Profa. Sandra Rejane Gomes Miessa
Reitora

Profa. Dra. Marília Ancona Lopez
Vice-Reitora de Graduação

Profa. Dra. Marina Ancona Lopez Soligo
Vice-Reitora de Pós-Graduação e Pesquisa

Profa. Dra. Claudia Meucci Andreatini
Vice-Reitora de Administração e Finanças

Profa. M. Marisa Regina Paixão
Vice-Reitora de Extensão

Prof. Fábio Romeu de Carvalho
Vice-Reitor de Planejamento

Prof. Marcus Vinícius Mathias
Vice-Reitor das Unidades Universitárias

Profa. Silvia Renata Gomes Miessa
Vice-Reitora de Recursos Humanos e de Pessoal

Profa. Laura Ancona Lee
Vice-Reitora de Relações Internacionais

Profa. Melânia Dalla Torre
Vice-Reitora de Assuntos da Comunidade Universitária

UNIP EaD

Profa. Elisabete Brihy
Profa. M. Isabel Cristina Satie Yoshida Tonetto

Material Didático

Comissão editorial:

Profa. Dra. Christiane Mazur Doi
Profa. Dra. Ronilda Ribeiro

Apoio:

Profa. Cláudia Regina Baptista
Profa. M. Deise Alcantara Carreiro
Profa. Ana Paula Tôrres de Novaes Menezes

Projeto gráfico:

Prof. Alexandre Ponzetto

Revisão:

Kleber Souza
Maria Cecília França

Sumário

Algoritmos e Estrutura de Dados em Python

APRESENTAÇÃO	7
INTRODUÇÃO	9

Unidade I

1 INTRODUÇÃO A ALGORITMOS, LINGUAGEM PYTHON E ESTRUTURAS DE DADOS	13
1.1 Conceitos fundamentais: o que são algoritmos e estruturas de dados e como utilizar a linguagem Python	15
1.2 Análise de complexidade: introdução à notação Big-O e à análise de eficiência	50
2 ESTRUTURAS DE DADOS LINEARES	55
2.1 Listas e arrays: manipulação, pesquisa, inserção e remoção	57
2.2 Pilhas e filas: implementação, operações e casos de uso	79

Unidade II

3 ESTRUTURAS DE DADOS NÃO LINEARES	104
3.1 Árvores: conceitos, árvores binárias e travessias (pré-ordem, em ordem e pós-ordem)	104
3.2 Grafos: representação (matriz de adjacência e listas de adjacência) e travessias (DFS e BFS)	128
4 ALGORITMOS DE ORDENAÇÃO	149
4.1 Ordenação simples: Bubble Sort, Selection Sort e Insertion Sort	151
4.2 Ordenação avançada: Merge Sort, Quick Sort e análise comparativa	162

Unidade III

5 ALGORITMOS DE PESQUISA	177
5.1 Pesquisas linear e binária: funcionamento e eficiência	178
5.2 Pesquisa em árvores: busca binária e introdução a árvores balanceadas (AVL e Red-Black)	185
6 ESTRUTURAS AVANÇADAS DE DADOS	194
6.1 Hash Tables: conceito, funções hash e resolução de colisões	195
6.2 Heaps: heaps binários e introdução ao Heap Sort	202

Unidade IV

7 ALGORITMOS EM GRAFOS.....	214
7.1 Caminhos mínimos: Dijkstra e Bellman-Ford	215
7.2 Grafos com árvores: algoritmos de Kruskal e Prim para árvores geradoras mínimas.....	227
8 APLICAÇÕES PRÁTICAS E PROBLEMAS CLÁSSICOS.....	238
8.1 Problemas de divisão e conquista: análise de exemplos como o problema da mochila.....	240
8.2 Programação dinâmica: introdução e resolução de problemas clássicos (como a subsequência comum mais longa).....	247

APRESENTAÇÃO

A computação contemporânea demanda cada vez mais soluções inteligentes, escaláveis e eficientes. Além de explicitar as bases em lógica de programação, esta disciplina aprofunda a capacidade de conceber e otimizar soluções computacionais, servindo como pilar fundamental para praticamente todas as áreas da tecnologia.

Um estudante de computação precisa aprender os fundamentos e as estruturas de dados para conseguir dominar uma linguagem de programação, porque essa combinação é fundamental para resolver problemas de maneira eficiente e eficaz. Estruturas de dados representam a maneira como as informações são organizadas, armazenadas e manipuladas em um programa, enquanto a linguagem de programação é a ferramenta para implementar essas ideias. Sem a compreensão sólida de estruturas de dados, o programador pode enfrentar sérias limitações no desenvolvimento de soluções otimizadas.

Um aluno que sabe programar em uma linguagem, mas não conhece algoritmos e estruturas de dados, é equivalente a um chef de cozinha que sabe usar panelas, facas e fogões, mas não entende os ingredientes e suas propriedades. Esse chef pode preparar pratos básicos ou seguir receitas simples, mas terá dificuldade em criar pratos complexos, equilibrar sabores ou adaptar receitas para diferentes restrições alimentares ou contextos culturais.

Por exemplo, assim como um chef precisa saber que ovos emulsionam e estabilizam molhos, ou que diferentes cortes de carne exigem técnicas de cozimento distintas, um programador precisa entender que uma pilha é útil para problemas que exigem rastreamento reverso (como desfazer ações em um editor) e que uma tabela hash oferece acessos rápidos a dados mapeados por chaves. Sem esse conhecimento profundo, o chef pode gastar muito tempo experimentando soluções improvisadas que não são as mais adequadas, resultando em pratos com qualidade abaixo do esperado. De forma semelhante, um programador sem conhecimento de estruturas de dados pode criar códigos que funcionam, mas com desempenho e eficiência longe do ideal. Ambos têm potencial, mas carecem das bases teórica e prática que elevam suas habilidades a um nível profissional.

Ao longo desta disciplina, os estudantes aprenderão a analisar e projetar algoritmos sob uma perspectiva de eficiência, com a introdução do conceito de complexidade e a notação Big-O. Com isso, torna-se possível comparar e escolher a melhor abordagem para cada tipo de problema, aspecto essencial em cenários reais, nos quais recursos computacionais são limitados e a velocidade de processamento pode ser decisiva.

A organização dos dados também ganha destaque, pois estruturas lineares (listas, pilhas e filas) e não lineares (árvores e grafos) são essenciais para armazenar e manipular informações de maneira estruturada. A partir dessa compreensão, o leitor será capaz de desenvolver programas mais robustos e bem organizados, assim como passará a dominar técnicas de pesquisa (linear, binária e em árvores) e ordenação (Bubble Sort, Merge Sort, Quick Sort, entre outros). Esses algoritmos, aparentemente simples, formam a base para o desenvolvimento de aplicações complexas em áreas como inteligência artificial, análise de grandes volumes de dados e criação de sistemas de recomendação.

O aprendizado da linguagem de programação (neste caso, conhecida como Python) e de estruturas de dados permite ao estudante compreender como organizar informações de forma lógica e eficiente, o que é essencial em problemas que exigem alto desempenho, como buscas rápidas, ordenações eficientes e manipulação de grandes volumes de dados. Por exemplo, entender a diferença entre uma pilha e uma lista encadeada, ou entre uma árvore binária e um grafo, pode ser decisivo para escolher a abordagem correta para resolver um problema específico.

Começaremos com os princípios básicos da programação. Na medida em que a disciplina avança, estruturas mais sofisticadas são exploradas – como tabelas de dispersão (hash tables) e heaps –, o que permite manipular dados de forma ainda mais eficiente. Também serão introduzidos algoritmos específicos para resolver problemas em grafos, incluindo caminhos mínimos (Dijkstra e Bellman-Ford) e árvores geradoras mínimas (Kruskal e Prim). Tais conceitos são amplamente utilizados em aplicações de redes, logística e rotas de transporte e em inúmeros outros domínios.

Por fim, as aplicações práticas englobam problemas clássicos de divisão e conquista e de programação dinâmica, treinando o raciocínio para reconhecer padrões de resolução eficientes. Essas técnicas permitem uma abordagem mais científica para a tomada de decisões ao solucionar problemas, tornando o desenvolvimento de software mais confiável e orientado a resultados.

Aprender algoritmos e estruturas de dados em Python também capacitará o estudante na utilização desses conceitos em outras linguagens de programação, uma vez que o conhecimento sobre algoritmos e estruturas de dados é, em grande parte, independente da linguagem. Estruturas de dados, como listas, filas, pilhas, árvores e grafos, representam conceitos fundamentais de computação que não dependem de uma implementação específica, mas de como os dados são organizados, armazenados e acessados.

Python é frequentemente usado como uma linguagem introdutória para aprender algoritmos e estruturas de dados devido à sua sintaxe simples e aos recursos integrados, como listas e dicionários, que tornam a implementação inicial mais acessível. No entanto, os conceitos aprendidos, como operações de inserção, remoção, busca e iteração, são transferíveis para qualquer linguagem.

Por exemplo, se uma pessoa aprende em Python que uma lista pode ser usada como uma pilha ao aplicar os métodos `append()` e `pop()`, ela será capaz de implementar o mesmo conceito na linguagem de programação C++, usando um `std::vector` ou na linguagem Java utilizando a classe `Stack`. Da mesma forma, a lógica de uma árvore binária de busca ou um algoritmo de busca em largura em grafos permanece a mesma, independentemente da linguagem. O que muda é a sintaxe e, às vezes, as bibliotecas ou os recursos oferecidos.

Além disso, algoritmos e estruturas de dados fazem parte da base teórica da computação e são amplamente exigidos em entrevistas de emprego e competições de programação. Dominar esses temas demonstra conhecimento técnico e habilidade de aplicar conceitos abstratos em situações práticas.

Em suma, a presente disciplina permite exercitar os conhecimentos sobre lógica de programação, promovendo a evolução do aluno rumo a competências de análise e à otimização cada vez mais apuradas. Esse amadurecimento é imprescindível para o sucesso em projetos de grande porte, garantindo o domínio de metodologias que sustentam o desenvolvimento de sistemas de alto desempenho e qualidade.

INTRODUÇÃO

No panorama atual da tecnologia, em que sistemas de informação são cada vez mais complexos e interligados, a compreensão profunda de algoritmos e estruturas de dados torna-se uma competência indispensável para quem deseja se destacar no mercado de trabalho e na área de computação.

Um dos conceitos de base que apresentaremos no tópico inicial deste livro-texto é a análise de complexidade, abordada por meio da notação Big-O. Essa análise possibilita mensurar o quão eficientemente um algoritmo se comporta à medida que o volume de dados ou a complexidade das tarefas aumenta.

Imagine, por exemplo, um cenário comum para muitos estudantes: organizar arquivos em pastas de um computador pessoal. Se o número de arquivos não for muito grande, pode-se simplesmente arrastar e soltar cada item manualmente, sem grandes problemas. Contudo, quando a quantidade de arquivos começa a crescer de maneira exponencial – como ocorre em grandes empresas ou plataformas de compartilhamento de documentos –, esse processo manual torna-se impraticável. Nesses casos, um algoritmo eficiente, capaz de classificar e organizar os arquivos automaticamente, faz toda a diferença.

A análise de complexidade nos ajuda a escolher métodos de ordenação ou de busca mais adequados, evitando que o tempo de processamento se torne inviável em cenários de grande escala. Tal abordagem permite que o aluno planeje e desenvolva soluções que sejam funcionais e capazes de lidar com quantidades substanciais de dados. O tópico 1 faz uma introdução à linguagem Python, com o objetivo de preparar o aluno para os próximos tópicos.

O estudo das estruturas de dados lineares, como listas, pilhas e filas, é outro elemento essencial no desenvolvimento de programas de computador e será objeto do tópico 2. Muitas vezes, o estudante já teve contato com listas simples em projetos iniciais, como criar uma pequena lista de tarefas a ser exibida na tela. No entanto, neste livro-texto, aprenderá a fundo como essas estruturas funcionam internamente e por que elas podem ser mais ou menos adequadas em certas situações. Um exemplo prático do dia a dia que ilustra o funcionamento de uma fila (queue) é a caixa de supermercado: cada novo cliente que chega se posiciona no final da fila, enquanto o primeiro cliente da fila é atendido e sai do sistema. Em termos computacionais, isso se traduz em operações de enfileiramento (enqueue) e de desenfileiramento (dequeue), cuja implementação impacta diretamente a eficiência de programas que processam tarefas em sequência, como impressão de documentos ou tratamento de requisições em servidores web.

Já as pilhas (stacks) seguem o princípio de LIFO (Last In, First Out), podendo ser exemplificadas por uma pilha de livros na mesa de estudos do aluno: o último livro colocado no topo é o primeiro a ser retirado quando o estudante quer consultá-lo novamente. Em computação, as pilhas são fundamentais para diversas rotinas internas, como gerenciamento de chamadas de função e de histórico de navegação em navegadores de internet. Entender como implementar essas estruturas e como elas operam em tempo de execução capacita o aluno a diagnosticar problemas e otimizar o código em uma variedade de cenários práticos.

Em seguida, no tópico 3, direcionaremos o aluno para o estudo de estruturas de dados não lineares, a começar pelas árvores. As árvores podem ser entendidas como representações hierárquicas, tal qual uma árvore genealógica ou a estrutura de pastas de um sistema operacional. Em ambos os casos, há um elemento raiz que gera diversos filhos, que, por sua vez, também podem gerar seus próprios filhos. Em um ambiente computacional, esse conceito é amplamente utilizado para a organização de dados em base de dados hierárquica, sistemas de arquivos e, de forma ainda mais avançada, em algoritmos de roteamento de redes. Também exploraremos o conceito de travessias em árvores (pré-ordem, em ordem e pós-ordem) para encontrar ou manipular dados de maneira sistemática, o que prepara o aluno para analisar aplicações como a montagem de expressões aritméticas e a manipulação de estruturas complexas, como árvores de sintaxe em compiladores.

Ainda sobre estruturas não lineares, os grafos desempenham papel de extrema importância em diversos campos. Um grafo pode ser visto como um conjunto de nós e conexões (arestas) entre esses nós e podem representar mapas de cidades, redes sociais ou até mesmo um sistema de distribuição de água em uma região. Essa versatilidade faz com que grafos sejam frequentemente estudados nas disciplinas de engenharia de software, análise de redes e ciência de dados e em muitos outros ramos que envolvem conexões e relacionamentos complexos. Para ilustrar de forma prática, basta pensar em aplicativos de navegação ou de entrega de comida: esses sistemas necessitam calcular rotas eficientes entre pontos geográficos, e esse cálculo internamente utiliza algoritmos de grafos como a BFS (Breadth-First Search) e DFS (Depth-First Search), além de algoritmos de caminho mínimo como Dijkstra ou Bellman-Ford, também apresentados neste livro-texto. Ao entender como esses algoritmos funcionam, o aluno adquire a capacidade de desenvolver soluções que vão desde jogos de computador até aplicações que envolvem logística e transporte em grande escala.

Além das estruturas, no tópico 4, estudaremos com profundidade os algoritmos de ordenação, que podem parecer, à primeira vista, puramente acadêmicos, mas têm aplicabilidade prática em quase todos os sistemas de computação que lidam com dados. Alguns exemplos bastante comuns são a apresentação de resultados de busca em sites de e-commerce ou a organização de contatos em redes sociais. Revisitaremos algoritmos simples, como Bubble Sort, Selection Sort e Insertion Sort; que, embora não sejam os mais eficientes para grandes volumes de dados, são fundamentais para a compreensão de como a comparação e a troca de elementos operam. Posteriormente, o aluno será introduzido a algoritmos mais sofisticados, como Merge Sort e Quick Sort, explorando suas vantagens e desvantagens e compreendendo como a análise de complexidade orienta a decisão de qual método aplicar em cada contexto. Essa visão crítica permite que o futuro profissional da computação adeque o algoritmo às necessidades específicas de desempenho e uso de recursos, essenciais em aplicações web e em sistemas de tempo real.

O tópico 5 também cobre algoritmos de pesquisa, tema essencial em bancos de dados, buscas em sistemas operacionais e pesquisa de registros em aplicações corporativas. Abordaremos a pesquisa linear, que, embora seja de fácil implementação, mostra-se ineficiente quando há necessidade de encontrar informações em um universo muito grande de dados. Para contornar essa limitação, introduziremos a pesquisa binária, que exige a listagem pré-ordenada dos elementos, mas pode localizá-los em um número significativamente menor de comparações. Na prática, essa busca binária é análoga ao que fazemos quando procuramos um verbete em um dicionário impresso: em vez de checar cada página, abrimos o livro próximo à letra desejada e, com alguns poucos saltos, chegamos ao tópico exato. Esse

tipo de analogia ajuda a compreender por que determinados algoritmos são preferíveis em certas condições de uso.

Para lidar com grandes volumes de dados de maneira eficiente, mostraremos, no tópico 6, estruturas mais avançadas, como as tabelas de dispersão (hash tables) e os heaps. As tabelas de dispersão são amplamente utilizadas em sistemas de cache, gestão de senhas e implementações de dicionários em linguagens de programação. Um exemplo cotidiano de hash é a organização de chaves em compartimentos na portaria de um prédio: cada apartamento pode ter sua gaveta de chaves e o porteiro dispersa essas chaves nessas gavetas de forma a encontrá-las rapidamente quando o morador retorna. Em termos computacionais, a função hash cumpre o papel de indicar, de maneira calculada, em qual espaço na tabela aquele elemento deve ser armazenado, tornando as operações de busca e inserção potencialmente muito rápidas.

Os heaps, por sua vez, são estruturas que facilitam a obtenção do maior ou do menor elemento de um conjunto em tempo otimizado. Tal característica é útil quando se trabalha com sistemas de agendamento de tarefas, em que é preciso sempre priorizar, por exemplo, as tarefas mais urgentes ou com maior relevância. O Heap Sort, um algoritmo de ordenação derivado do conceito de heap, exemplifica como a estrutura pode ser utilizada para organizar dados sem perder o controle sobre a eficiência das operações.

No tópico 7, exploraremos mais a fundo os algoritmos voltados para grafos, com ênfase em rotas ótimas e árvores geradoras mínimas. É nesse momento que o aluno se depara com problemas como encontrar o caminho mais curto entre dois pontos (Dijkstra e Bellman-Ford) ou identificar uma rede de conexões que use o menor número possível de cabos para interligar todos os nós (Kruskal e Prim). Esses conceitos são cruciais em aplicações como planejamento de rotas de distribuição de mercadorias, redes de telecomunicações e até mesmo na determinação de percursos dentro de jogos digitais.

Por último, o tópico derradeiro oferece oportunidades de aplicação prática e de resolução de problemas clássicos por meio de técnicas de divisão e conquista e de programação dinâmica. Em divisão e conquista, o aluno aprenderá a quebrar um problema em partes menores, resolvê-las individualmente e combinar essas soluções para o todo. Já na programação dinâmica, valorizaremos a ideia de reaproveitar resultados parciais para calcular soluções de problemas mais complexos, tornando o processo muito mais rápido. Situações cotidianas que ilustram essas técnicas incluem planejar a melhor maneira de estudar para diversas provas (dividindo cada matéria em pequenos blocos de conteúdo) ou ainda calcular a rota mais econômica de uma viagem, considerando custos de transporte e hospedagem em cada trecho. A famosa mochila de viagem (problema da mochila) pode ser estudada sob a ótica da programação dinâmica, na qual se busca a melhor combinação de itens que possam ser levados sem exceder uma capacidade limite, maximizando-se o valor ou a utilidade do que é transportado.

Toda essa gama de conceitos preparará o leitor para lidar com uma ampla variedade de desafios em disciplinas futuras e projetos reais de desenvolvimento de software. As noções de eficiência computacional, organização lógica dos dados e seleção criteriosa de algoritmos reverberam em áreas como ciência de dados, desenvolvimento web, inteligência artificial e sistemas embarcados. Nesse sentido, a dedicação à leitura deste livro-texto é muito mais do que um requisito curricular – ela é a

base que sustenta o pensamento crítico e a capacidade de inovação de todo futuro profissional da área de tecnologia da informação.

Por meio de exemplos práticos e de problemas extraídos do cotidiano, o aluno encontrará motivação e sentido para cada estrutura estudada e algoritmo implementado, visualizando como esses conceitos podem melhorar diretamente aplicativos que utilizamos todos os dias, seja em casa, no trabalho ou na esfera acadêmica. Por esse motivo, a prática da programação é imprescindível. O leitor pode simplesmente repetir os exemplos deste material, mas também pode adaptá-los, modificá-los e expandi-los.

Desejamos uma ótima leitura e um excelente treino!

Unidade I

1 INTRODUÇÃO A ALGORITMOS, LINGUAGEM PYTHON E ESTRUTURAS DE DADOS

Algoritmos e estruturas de dados são pilares fundamentais da ciência da computação. A linguagem Python, conhecida por sua clareza e simplicidade, é frequentemente utilizada como ferramenta para introduzir esses conceitos.

Um algoritmo pode ser classificado como uma sequência finita de passos bem definidos, projetados para resolver um problema ou executar uma tarefa específica. Essa definição abrange desde operações matemáticas simples, como somar dois números, até processos mais complexos, como ordenar grandes volumes de dados ou realizar buscas em estruturas de informação. As estruturas de dados representam maneiras de organizar, armazenar e manipular os dados necessários para que algoritmos sejam executados de maneira eficiente.

O entendimento e a aplicação desses dois conceitos são indispensáveis para qualquer programador ou cientista da computação, pois constituem a base para a resolução de problemas computacionais de forma eficaz.

A relevância dos algoritmos está relacionada à sua capacidade de automatizar processos, reduzindo o esforço humano necessário para realizar tarefas repetitivas ou complicadas. Eles desempenham papel crítico em praticamente todos os campos da tecnologia moderna, desde o aprendizado de máquina e a inteligência artificial até a gestão de bancos de dados e os sistemas operacionais. A escolha de algoritmos adequados pode significar a diferença entre uma aplicação funcional e eficiente e outra que consome recursos desnecessários ou apresenta baixo desempenho.

Estruturas de dados complementam esse processo ao fornecer os meios para que informações sejam organizadas e acessadas de maneira eficiente. Com a estrutura correta, operações como busca, inserção, remoção ou ordenação de dados podem ser realizadas com maior rapidez, com economia de tempo e recursos computacionais.

A análise da eficiência de algoritmos e estruturas de dados é outra peça-chave na compreensão do funcionamento de sistemas computacionais. É nesta etapa que entra o conceito de complexidade computacional, o qual ajuda a quantificar e prever o desempenho de um algoritmo em termos de tempo de execução e de uso de memória. A notação Big-O é um dos instrumentos mais utilizados para descrever a complexidade de algoritmos, pois oferece uma forma padronizada de expressar o comportamento de uma solução conforme o tamanho do conjunto de dados aumenta. Essa análise é essencial, especialmente em aplicações que lidam com grandes volumes de dados, nos quais pequenas diferenças na eficiência podem ter impactos significativos.

No contexto da linguagem Python, a compreensão de algoritmos e estruturas de dados ganha ainda mais importância devido à ampla gama de bibliotecas e de recursos que a linguagem oferece. Embora Python seja frequentemente reconhecida por sua facilidade de uso e simplicidade, esses mesmos atributos podem levar à percepção equivocada de que o desempenho é um fator secundário. No entanto, em muitos casos, a escolha errada de um algoritmo ou de uma estrutura de dados pode comprometer a funcionalidade de um programa, mesmo em Python. Por isso, inclusive desenvolvedores que trabalham com linguagens de alto nível como Python devem investir no estudo desses fundamentos.

A notação Big-O, que já foi mencionada, é uma ferramenta essencial para descrever o comportamento de algoritmos em termos de complexidade de tempo e espaço. Ela utiliza símbolos matemáticos para expressar como o desempenho de um algoritmo escala em relação ao tamanho do conjunto de entrada. Por exemplo, $O(1)$ indica que o tempo de execução é constante, independentemente do tamanho da entrada. Essa é uma característica de operações como acessar diretamente um elemento em uma lista pelo seu índice. $O(n)$, por outro lado, indica que o tempo de execução aumenta linearmente com o tamanho da entrada, como ocorre em uma busca linear. Conforme os problemas se tornam mais complexos, aparecem outras notações, como $O(n \log n)$, $O(n^2)$ e $O(2^n)$, cada uma refletindo diferentes níveis de eficiência e escalabilidade.

A análise de complexidade é especialmente importante quando se trabalha com algoritmos que serão utilizados em sistemas de grande escala ou em tempo real. Por exemplo, algoritmos de ordenação ou de busca que funcionam bem em pequenas coleções de dados podem se tornar inviáveis em aplicações como motores de busca ou sistemas financeiros, nos quais milhões de operações precisam ser realizadas em frações de segundo. Além disso, a eficiência não é apenas uma questão de desempenho percebido pelo usuário, mas de custo operacional. Reduzir o consumo de recursos computacionais pode significar economias substanciais em ambientes corporativos, nos quais servidores e infraestruturas em nuvem são cobrados com base no uso.

É interessante notar que Python fornece várias estruturas de dados prontas, como listas, tuplas, dicionários e conjuntos, que podem ser usadas para implementar algoritmos de forma eficiente. Além disso, bibliotecas como NumPy, pandas e scikit-learn oferecem recursos adicionais que simplificam ainda mais a manipulação de dados e a implementação de soluções.

No entanto, a facilidade de uso dessas ferramentas não elimina a necessidade de compreender os fundamentos. Muitos desafios computacionais não podem ser resolvidos simplesmente utilizando ferramentas prontas, exigindo que os desenvolvedores compreendam os algoritmos subjacentes e sejam capazes de adaptá-los ou desenvolvê-los do zero quando necessário.

Neste tópico, estudaremos em detalhes tais conceitos, fundamentais para a formação de bons profissionais na área da computação.

1.1 Conceitos fundamentais: o que são algoritmos e estruturas de dados e como utilizar a linguagem Python

Um algoritmo, em termos simples, pode ser comparado a uma receita de cozinha. Assim como uma receita descreve, passo a passo, o que deve ser feito para preparar um prato, um algoritmo apresenta uma sequência clara e ordenada de instruções para resolver um problema ou realizar uma tarefa. Esse conceito não é exclusivo do mundo da tecnologia e pode ser encontrado em diversas situações diárias. Por exemplo, quando alguém explica como chegar a um local desconhecido, descrevendo o caminho em etapas, está, na prática, criando um algoritmo.

No contexto da computação, um algoritmo é um conjunto de instruções que um computador pode seguir para realizar uma operação específica. Ele funciona como uma espécie de plano ou manual para o processamento de informações. Imagine, por exemplo, que você deseja que um computador organize uma lista de nomes em ordem alfabética. Para realizar essa tarefa, ele precisa de um algoritmo que descreva como comparar os nomes, trocá-los de posição quando necessário e repetir o processo até que a lista esteja ordenada.

As características essenciais de um algoritmo são a clareza e a objetividade. Cada etapa deve ser bem definida, sem espaço para interpretações ambíguas. Isso é importante porque os computadores, diferentemente dos humanos, não são capazes de adivinhar intenções nem de interpretar informações implícitas. Um algoritmo eficiente garante que o computador execute exatamente o que se espera, de maneira previsível e confiável.

Os algoritmos estão presentes em quase tudo que envolve tecnologia. Desde o funcionamento de aplicativos no celular até as buscas realizadas na internet, eles desempenham papel fundamental. Quando você faz uma pesquisa online, por exemplo, um algoritmo é usado para encontrar e classificar as informações mais relevantes. Da mesma forma, quando um aplicativo sugere músicas ou filmes com base no que você já ouviu ou assistiu, ele está utilizando algoritmos para analisar padrões e tomar decisões.

Embora o conceito de algoritmo possa parecer abstrato à primeira vista, ele é essencialmente uma ferramenta prática e universal. Qualquer problema que possa ser resolvido seguindo etapas lógicas pode ser abordado com a criação de um algoritmo, o que torna esse conceito acessível e aplicável para qualquer pessoa, independentemente de seu nível de familiaridade com tecnologia.

De maneira mais técnica, um algoritmo pode ser definido como uma sequência finita, ordenada e bem definida de instruções que, ao serem executadas, resolvem um problema específico ou realizam uma tarefa. Cada passo do algoritmo deve ser claro, preciso e executável, garantindo que, para qualquer entrada válida, ele produza uma saída correspondente no final de sua execução. Essa abordagem fundamenta-se em princípios de lógica matemática e de estruturação computacional, sendo o coração de qualquer sistema que processa informações.

Formalmente, um algoritmo deve atender a certas propriedades, explicadas a seguir.

- **Finitude:** um algoritmo deve sempre terminar após um número finito de passos. Ele não pode continuar indefinidamente, a menos que seja projetado especificamente para isso (como em loops contínuos controlados por condições externas).
- **Precisão:** cada passo de um algoritmo precisa ser bem definido, ou seja, sem ambiguidades. O computador (ou o executor humano) deve ser capaz de compreender e executar cada etapa exatamente como descrita.
- **Entrada:** um algoritmo recebe zero ou mais entradas, que são os dados iniciais necessários para começar o processamento.
- **Saída:** um algoritmo produz uma ou mais saídas, que são os resultados esperados após a execução das instruções.
- **Eficácia:** cada instrução do algoritmo deve ser suficientemente básica para que possa ser executada em tempo finito, utilizando recursos computacionais disponíveis.

Em termos computacionais, um algoritmo pode ser implementado usando linguagens de programação como Python, Java, C++, entre outras. Ele é geralmente descrito em forma de pseudocódigo ou de fluxograma antes de ser traduzido para código executável, o que facilita sua compreensão e sua análise.

Para ilustrar, considere o problema clássico de determinar se um número inteiro é par ou ímpar. Um algoritmo técnico para resolver esse problema pode ser descrito assim como segue.

- Receber um número inteiro como entrada.
- Dividir o número por 2 e calcular o resto (operação de módulo).
- Se o resto for igual a 0, o número será par.
- Caso contrário, o número será ímpar.
- Retornar o resultado (par ou ímpar).

O pseudocódigo desse algoritmo seria como vemos a seguir.

```
Entrada: número inteiro `n`  
Se (n % 2 == 0):  
    Retornar "Par"  
Senão:  
    Retornar "Ímpar"
```

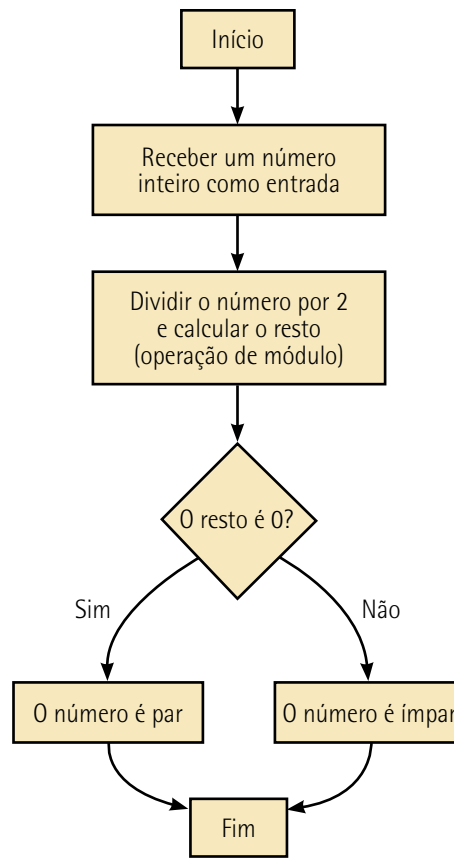



Figura 1 – Representação esquemática de um algoritmo que determina se um número inteiro é par ou ímpar

Esse exemplo simples mostra como um algoritmo descreve, de forma clara e estruturada, a solução de um problema específico, como podemos ver no fluxograma da figura anterior. Além disso, ele ilustra a importância de elementos como entrada, saída e lógica sequencial.

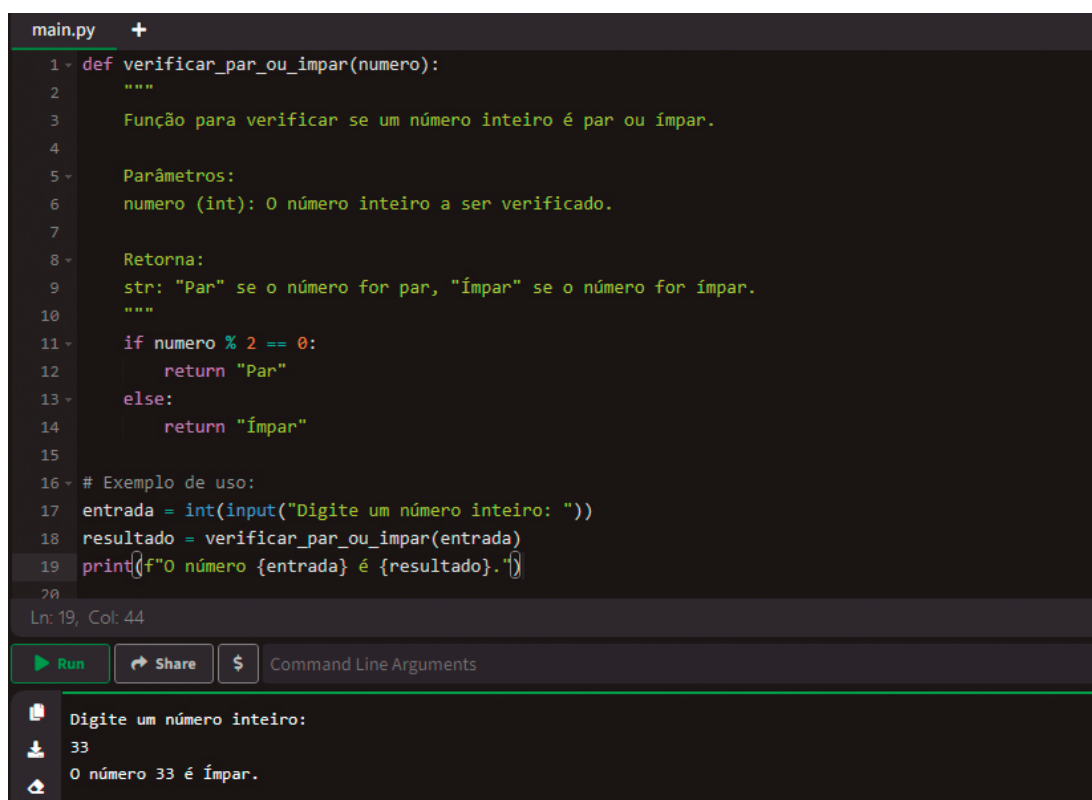
Uma análise mais técnica também inclui a eficiência do algoritmo, que é avaliada em termos de sua complexidade de tempo e espaço. Para o exemplo anterior, a complexidade de tempo é $O(1)$, porque a operação de módulo é executada em tempo constante, independentemente do tamanho da entrada. Em outras palavras, independentemente do número inteiro fornecido, seja ele pequeno como 2 ou gigantesco como 10 bilhões, o algoritmo executa sempre um número fixo e determinado de operações, sem variar conforme a entrada cresce. De fato, é verdade que, em uma execução prática, verificar se o número 2 é par será ligeiramente mais rápido do que verificar se o número 10 bilhões é par. Entretanto, essa diferença, no contexto da complexidade computacional, não é relevante ou significativa o suficiente para alterar a classificação.

A complexidade $O(1)$ não significa que o algoritmo levará exatamente o mesmo número de nanossegundos ou ciclos do processador independentemente do tamanho exato da entrada. Em vez disso, o significado real é que o número de passos fundamentais do algoritmo não aumenta conforme a entrada cresce. É o número de operações básicas, como somar, dividir ou testar uma condição simples, que permanece constante.

No exemplo apresentado, ocorre apenas uma operação básica essencial, que é a operação matemática do tipo "resto da divisão por 2". Essa operação é realizada diretamente pelo processador, com poucos ciclos internos, e independentemente de quão grande o número seja, não é preciso repetir várias vezes esse passo. A operação é feita uma única vez, apesar do valor da entrada.

A complexidade de espaço também é $O(1)$, pois nenhuma estrutura adicional é utilizada além do próprio número de entrada. Explicaremos essa notação com mais detalhes posteriormente. Essa análise permite identificar se o algoritmo é adequado para aplicações em que recursos computacionais precisam ser otimizados.

O código em Python para resolver esse problema é simples e direto, utilizando a operação de módulo (%) para determinar o resto da divisão por 2. A figura a seguir ilustra esse código-fonte, bem como um exemplo de execução do código usando o ambiente Online Python.



```
main.py +
1 def verificar_par_ou_impar(numero):
2     """
3     Função para verificar se um número inteiro é par ou ímpar.
4
5     Parâmetros:
6     numero (int): O número inteiro a ser verificado.
7
8     Retorna:
9     str: "Par" se o número for par, "Ímpar" se o número for ímpar.
10    """
11    if numero % 2 == 0:
12        return "Par"
13    else:
14        return "Ímpar"
15
16 # Exemplo de uso:
17 entrada = int(input("Digite um número inteiro: "))
18 resultado = verificar_par_ou_impar(entrada)
19 print(f"O número {entrada} é {resultado}.")
20
Ln: 19, Col: 44
Run Share $ Command Line Arguments
Digite um número inteiro:
33
O número 33 é ímpar.
```

Figura 2 – Ambiente Online Python (<https://shre.ink/xx4l>). Parte superior: código Python que implementa o algoritmo para determinar se um número inteiro é par ou ímpar. Parte inferior: exemplo de execução do código após pressionar o botão Run e digitar "33" e a tecla Enter

Python é uma linguagem de programação de alto nível, amplamente utilizada devido à sua simplicidade e legibilidade, o que a torna ideal tanto para iniciantes quanto para desenvolvedores experientes. Criada por Guido van Rossum na década de 1990, Python é usada em diversas áreas, como desenvolvimento web, ciência de dados, inteligência artificial e automação de tarefas. Uma característica fundamental dessa linguagem é que ela é interpretada, o que significa que seu código não precisa ser compilado antes de ser executado. Em vez disso, um interpretador (no exemplo da figura 2, usamos o

interpretador disponível em: <https://shre.ink/xx4d>) lê e executa o código linha por linha, permitindo um desenvolvimento mais ágil e facilitando a identificação de erros. Essa abordagem também torna Python altamente portátil, podendo ser executada em diferentes sistemas operacionais sem necessidade de ajustes significativos.

Vamos explicar o programa apresentado na figura 2. Não se preocupe se não entender em detalhes a explicação, você se familiarizará com os conceitos em breve. O programa inicia-se com a definição de uma função chamada `verificar_par_ou_imp`, cujo propósito consiste em avaliar se um número inteiro fornecido é par ou ímpar. Em Python, funções são blocos (trechos) de código que podem receber dados de entrada, processá-los e devolver um resultado. No exemplo, a função recebe um único parâmetro batizado de `numero` (vide linha 1), que deve ser um valor inteiro, e retorna uma cadeia de caracteres (`Par` ou `Ímpar`), indicando o resultado da verificação (linhas 12 e 14, respectivamente).

Logo após a declaração da função encontra-se um trecho de texto entre três aspas triplas (linhas 2 até 10). Essa seção constitui a documentação interna, denominada `docstring`, na qual se descreve o objetivo da função, as características do parâmetro de entrada e o tipo de valor que será devolvido. Embora não afete diretamente a execução do programa, essa prática facilita o entendimento do código por outras pessoas ou por ferramentas de desenvolvimento.

A lógica central reside no comando condicional `if`. Na Python, a expressão `numero % 2` calcula o resto da divisão do valor armazenado em `numero` por 2. Quando esse resto é igual a zero, significa que o número é divisível por 2, caracterizando-o como par. A instrução `if numero % 2 == 0` verifica essa condição. Se ela for verdadeira, a função retorna a cadeia de caracteres `Par` (linha 12); caso contrário, o bloco presente após a palavra-chave `else` será executado, retornando `Ímpar` (linha 14).

Encerrada a definição da função (linha 15), o programa procede à interação com o usuário. A linha que contém `input` solicita ao usuário que digite um número inteiro, exibindo na tela a mensagem `Digite um número inteiro:`. A função `input` (linha 17) sempre captura o texto digitado, retornando-o como uma cadeia de caracteres (`string`). Para converter essa `string` em um número inteiro, utiliza-se a função `int`, cujos parênteses envolvem o resultado de `input`. O valor convertido é atribuído à variável `entrada`.

Em seguida, ocorre a invocação da função `verificar_par_ou_imp`, passando-se como argumento o valor armazenado em `entrada`. O retorno da função, correspondendo a `Par` ou `Ímpar`, é armazenado na variável `resultado`. Por fim, o programa exibe na tela uma mensagem composta dinamicamente por meio de uma f-string: `f'O número {entrada} é {resultado}.'`. As chaves dentro da f-string servem para inserir, diretamente no texto impresso, os conteúdos das variáveis `entrada` e `resultado`, informando de forma clara e legível ao usuário o resultado da verificação.

Em resumo, o código demonstra três conceitos fundamentais de qualquer linguagem de programação: a criação de funções para organizar e reutilizar lógica, o uso de estruturas condicionais para tomar decisões com base em valores e a interação com o usuário por meio de entrada e saída de dados.

Abordaremos a linguagem Python com mais detalhes. Ela utiliza variáveis para armazenar valores sob nomes simbólicos, de forma a tornar o código mais legível e organizado. Uma variável é criada atribuindo-se um valor a um identificador, conforme o exemplo a seguir:

```
idade = 25
mensagem = "Bem-vindo ao curso de Python"
```

Na primeira linha, o nome `idade` recebe (armazena) o número inteiro 25; na segunda, o nome `mensagem` armazena uma sequência de caracteres (texto). Essa flexibilidade caracteriza a tipagem dinâmica: não é necessário declarar antecipadamente o tipo de dado, pois Python identifica automaticamente se a variável conterá números, textos, listas ou outros objetos.

Em qualquer ponto do programa, o valor vinculado a uma variável pode ser alterado por meio de uma nova atribuição. Por exemplo, ao executar `idade = idade + 1`, o valor originalmente atribuído à `idade` é substituído por outro incrementado em uma unidade. Essa prática habilita a criação de contadores, acumuladores e demais mecanismos de controle numérico que surgem em tarefas cotidianas de programação.

Neste ponto, convém detalhar também o conceito de função, que é um bloco de código nomeado, responsável por encapsular operações reutilizáveis e separar responsabilidades. A definição de uma função inicia-se com a palavra-chave `def`, seguida do nome desejado e de parênteses contendo eventuais parâmetros. O corpo da função, recuado em relação à margem, contém instruções que serão executadas sempre que o bloco for invocado. No exemplo a seguir, a função realiza a soma de dois valores:

```
def somar(a, b):
    resultado = a + b
    return resultado
```

Ao solicitar a soma de 3 e 5, o programa precisa invocar `somar(3, 5)` e recebe de volta o valor 8. A palavra-chave `return` indica qual valor será devolvido ao ponto de chamada, permitindo o uso dessa saída em novos cálculos ou apresentações.

É possível definir funções sem retorno explícito, quando o interesse consiste em executar uma ação específica, tal como exibir uma mensagem. No fragmento a seguir, a função imprime cumprimento personalizado:

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

Nesse caso, ao chamar `cumprimentar("Mariana")`, o programa imprime `Olá, Mariana! Seja bem-vindo.` e prossegue sem gerar valor de retorno.

Variáveis e funções trabalham em conjunto para estruturar programas que permaneçam claros e fáceis de modificar. Por exemplo, ao calcular a área de um retângulo, pode-se armazenar medidas em variáveis e delegar o cálculo a uma função:

```
largura = 7
altura = 4

def calcular_area(larg, alt):
    return larg * alt

area = calcular_area(largura, altura)
print("Área do retângulo:", area)
```

Dessa forma, o valor de `area` resulta da operação realizada internamente à função, permitindo reutilizar `calcular_area` sempre que se desejar obter área de novos retângulos sem copiar e colar lógica.

Em síntese, variáveis fornecem meios de nomear e manipular dados ao longo da execução, enquanto funções promovem a organização do código em módulos concisos e reutilizáveis. Esses conceitos formam a base de praticamente qualquer programa em Python, favorecendo a manutenção e a expansão de projetos desde pequenos scripts até aplicações complexas.

A linguagem também oferece mecanismos que permitem ao programa tomar decisões e repetir ações de maneira controlada, conferindo flexibilidade e clareza ao desenvolvimento. As estruturas de decisão verificam condições e direcionam o fluxo de execução para diferentes trechos de código, enquanto as estruturas de controle de repetição permitem a execução contínua de um bloco até que uma condição seja satisfeita.

Para ilustrar a primeira categoria, a instrução `if` funciona como um ponto de verificação: ela avalia uma expressão `e`, caso o resultado seja verdadeiro, executa um bloco específico. No exemplo a seguir, o programa analisa se um valor numérico é maior que 10 e informa ao usuário:

```
numero = 7
if numero > 10:
    print("Valor maior que dez")
```

Nesse caso, como a variável `numero` contém 7, inferior a 10, o conteúdo do bloco (que contém a instrução `print`) não é executado. Quando se deseja oferecer um caminho alternativo, utiliza-se `else`, o qual executa o bloco vinculado caso a condição inicial seja falsa:

```
numero = 7
if numero > 10:
    print("Valor maior que dez")
else:
    print("Valor dez ou menor")
```

A inclusão de `elif` (abreviação de `else if`) possibilita múltiplas verificações sequenciais, cada uma com sua expressão. No exemplo a seguir, o programa classifica uma temperatura em três níveis:

```
temperatura = 23
if temperatura > 30:
    print("Clima quente")
```

```
elif temperatura > 20:
    print("Clima ameno")
else:
    print("Clima frio")
```

A repetição de tarefas é tratada por meio de laços de controle. O laço `while` repete um bloco enquanto uma condição permanecer verdadeira. No trecho a seguir, o código solicita ao usuário que digite a palavra `sair` para interromper o processo:

```
comando = ""
while comando != "sair":
    comando = input("Digite 'sair' para encerrar: ")
print("Programa encerrado")
```

Observa-se que, enquanto `comando` for diferente de `sair`, o laço continua solicitando entrada. Quando a condição se torna falsa, o programa prossegue para a instrução seguinte ao bloco.

Outra forma de repetição consiste no laço `for`, apropriado para percorrer sequências predefinidas, como listas ou intervalos de números. O exemplo a seguir exibe os cinco primeiros números naturais:

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

A expressão `range(5)` cria uma série de valores de 0 a 4, sobre os quais o laço itera automaticamente, atribuindo a cada passo o valor à variável `i`. A cada repetição, o bloco interno é executado até o final da sequência.

Você deve ter observado que a linguagem Python adota a indentação como elemento central para definir a estrutura do programa, substituindo o uso de chaves ou palavras-chave mais verbosas presentes em outras linguagens. Cada bloco de código – como o corpo de uma função, de um laço ou de uma condição – deve ser recuado em relação à margem. Essa convenção torna o código mais legível, pois torna visível o agrupamento lógico das instruções.

Por exemplo, ao declarar uma função que imprime uma mensagem, o recuo de quatro espaços – padrão amplamente adotado pela comunidade – sinaliza o início do bloco pertencente à definição:

```
def cumprimentar(nome):
    print(f"Olá, {nome}!")
```

Nesse trecho, a linha que começa com `print` está avançada em quatro espaços, indicando-se como parte do corpo de `cumprimentar`. Ao retornar o fluxo para a margem original, conclui-se o bloco.

Quando se utiliza um laço `for`, a mesma convenção de indentação aplica-se para agrupar as instruções que serão repetidas:

```
for i in range(3):  
    print("Passo", i)  
print("Laço concluído")
```

A primeira linha após `for` segue o recuo estipulado, caracterizando-se como pertencente ao laço. A instrução `print("Laço concluído")`, sem recuo, permanece fora do bloco repetitivo.

A observância rigorosa da indentação é imprescindível: qualquer desalinhamento gera erro de sintaxe, impedindo a execução do programa. Em caso de recuos inconsistentes, Python sinaliza `IndentationError`, indicando que foi encontrada uma posição inesperada. Dessa forma, essa convenção promove clareza e evita ambiguidade na associação de comandos, sendo considerada um dos traços distintivos da legibilidade do código Python.

A seguir, faremos uma série de exercícios práticos para que você se familiarize com a linguagem Python. Cada exercício trata de um cenário de aplicação real no qual aspectos básicos da linguagem serão aprofundados. É importante praticar cada um deles seguindo as instruções fornecidas.

Exercício 1. Controle de gastos mensais

Imagine que você está desenvolvendo um sistema simples para ajudar as pessoas a calcularem seus gastos mensais e analisarem seu orçamento. O programa deve receber informações do usuário sobre seus rendimentos e suas despesas e, com base nisso, oferecer recomendações. Crie um programa em Python que atenda aos seguintes requisitos listados a seguir.

• Entrada de dados

- Solicitar que o usuário insira seu rendimento mensal (salário ou outras fontes de renda).
- Requisitar que o usuário insira suas principais despesas mensais (como aluguel, alimentação, transporte e lazer).
- Perguntar ao usuário se ele tem um plano de poupança (sim ou não).

• Processamento

- Calcular o total de despesas e o saldo restante do rendimento.
- Se o saldo for negativo, exibir uma mensagem de alerta.
- Se o saldo for positivo e o usuário informou que tem plano de poupança, sugerir um valor a ser poupado (20% do saldo restante).
- Se o saldo for positivo e o usuário informou que não tem plano de poupança, recomendar que comece a poupar.

- **Estruturas de decisão**

- Usar `if`, `elif` e `else` para tratar diferentes cenários.
- Incluir condições para verificar se o usuário inseriu valores válidos.

- **Criação de funções**

- Criar uma função chamada `calcular_saldo` que receba os rendimentos e despesas como parâmetros e retorne o saldo.
- Criar uma função chamada `sugerir_poupanca` que receba o saldo como parâmetro e calcule 20% dele.

- **Técnicas de depuração**

- Incluir instruções de `print` para exibir as variáveis durante a execução, ajudando a identificar possíveis erros no cálculo.
- Tratar possíveis entradas inválidas, como valores não numéricos ou negativos.

- **Instruções para o usuário**

- Executar o programa.
- Inserir os valores conforme solicitado.
- Verificar os cálculos e as mensagens exibidas.
- Experimentar inserir valores inválidos (como texto ou números negativos) para testar os tratamentos de erro.

Esse exercício incentiva a prática de estruturas de decisão, manipulação de variáveis, entrada de dados e técnicas básicas de depuração, além de explorar a criação de funções para modularizar o código.

Quando trabalhamos com estruturas de decisão, usamos `if`, `elif` e `else` de forma direta e quase natural.

Em Python, a manipulação de variáveis é dinâmica, o que significa que não precisamos declarar tipos antecipadamente; basta atribuir um valor e o interpretador define se é uma `string`, um número ou outro tipo, facilitando a criação e o teste de ideias sem burocracias.

A entrada de dados, realizada por meio da função `input()`, torna a interação com o usuário simples e direta, capturando informações que podem ser convertidas e validadas conforme a necessidade do programa.

Quanto às técnicas básicas de depuração, Python exibe mensagens de erro de forma clara, ajudando o desenvolvedor a identificar rapidamente onde o código não está se comportando como esperado. Também é possível usar a função `print()` para exibição de conteúdo das variáveis.

Por fim, a criação de funções utilizando `def` é uma prática que reforça a modularização do código, permitindo que cada parte do programa seja responsável por uma tarefa específica. Essa abordagem deixa o código mais organizado e legível – seguindo a convenção `snake_case` para nomes – e facilita a manutenção, bem como o reuso de código em projetos maiores.

A seguir, temos o código-fonte que soluciona o exercício.

```
def calcular_saldo(rendimento, despesas):
    return rendimento - despesas

def sugerir_poupanca(saldo):
    return saldo * 0.2

# Entrada de dados
try:
    rendimento = float(input("Digite seu rendimento mensal (em reais): "))
    if rendimento < 0:
        raise ValueError("O rendimento não pode ser negativo.")

    despesas = float(input("Digite o total de suas despesas mensais (em reais): "))
    if despesas < 0:
        raise ValueError("As despesas não podem ser negativas.")

    tem_poupanca = input("Você tem um plano de poupança? (sim/não): ").strip().
lower()
    if tem_poupanca not in ["sim", "não"]:
        raise ValueError("Resposta inválida. Digite 'sim' ou 'não'.")

    # Processamento
    saldo = calcular_saldo(rendimento, despesas)
    print(f"\nSeu saldo mensal é: R$ {saldo:.2f}")

    if saldo < 0:
        print("Atenção! Suas despesas estão maiores que seus rendimentos.")
    elif saldo > 0:
        if tem_poupanca == "sim":
            poupanca_sugerida = sugerir_poupanca(saldo)
            print(f"Recomendamos poupar R$ {poupanca_sugerida:.2f} este mês.")
        else:
            print("Considere criar um plano de poupança para aproveitar seu saldo positivo.")
    else:
        print("Seu orçamento está equilibrado, sem saldo restante para poupar.")

except ValueError as e:
    print(f"Erro na entrada de dados: {e}")
```

O funcionamento do código inicia com a definição de duas funções principais. A primeira, `calcular_saldo`, recebe dois argumentos, `rendimento` e `despesas`, que representam os ganhos e os gastos mensais, respectivamente. Ela retorna o `saldo`, que é a diferença entre o rendimento e as despesas. A segunda função, `sugerir_poupanca`, calcula um valor recomendado para poupar, considerando que 20% do saldo positivo seja destinado à poupança.

A etapa seguinte do código envolve a entrada de dados pelo usuário. Ela solicita três informações: o rendimento mensal, o total das despesas e se o usuário tem ou não um plano de poupança. Essas informações são obtidas usando a função `input`, que coleta os dados como texto, sendo necessário convertê-los para o tipo apropriado.

Para o rendimento e as despesas, a conversão é feita para números de ponto flutuante (`float`), já que os valores monetários podem incluir centavos. Além disso, verifica-se se os valores são não negativos, pois rendimentos ou despesas negativas não fazem sentido prático nesse contexto. Caso um valor negativo seja informado, o programa gera um erro do tipo `ValueError`, interrompendo a execução e exibindo uma mensagem explicativa ao usuário.

Na terceira entrada, o programa solicita se o usuário tem um plano de poupança, aceitando como respostas válidas apenas sim ou não. A resposta é convertida para letras minúsculas e espaços extras são removidos, a fim de garantir consistência. Se o usuário digitar algo diferente de sim ou não, um erro é gerado, informando que a entrada é inválida.

Após coletar e validar os dados, o programa realiza o cálculo do saldo utilizando a função `calcular_saldo`. O saldo resultante é exibido ao usuário com duas casas decimais, indicando a diferença entre os rendimentos e as despesas. O código, então, avalia a situação financeira.

- Caso o saldo seja negativo, o programa alerta o usuário de que as despesas superaram os rendimentos, sugerindo atenção ao orçamento.
- Se o saldo for positivo, o programa verifica se o usuário tem um plano de poupança. Para quem já tem poupança, o valor recomendado para ser poupado é calculado pela função `sugerir_poupanca` e exibido ao usuário. Se o usuário não tiver um plano de poupança, o programa sugere considerar a criação de um, aproveitando o saldo positivo.
- Para o caso em que o saldo é exatamente zero, o programa informa que o orçamento está equilibrado, não havendo sobra para poupança.

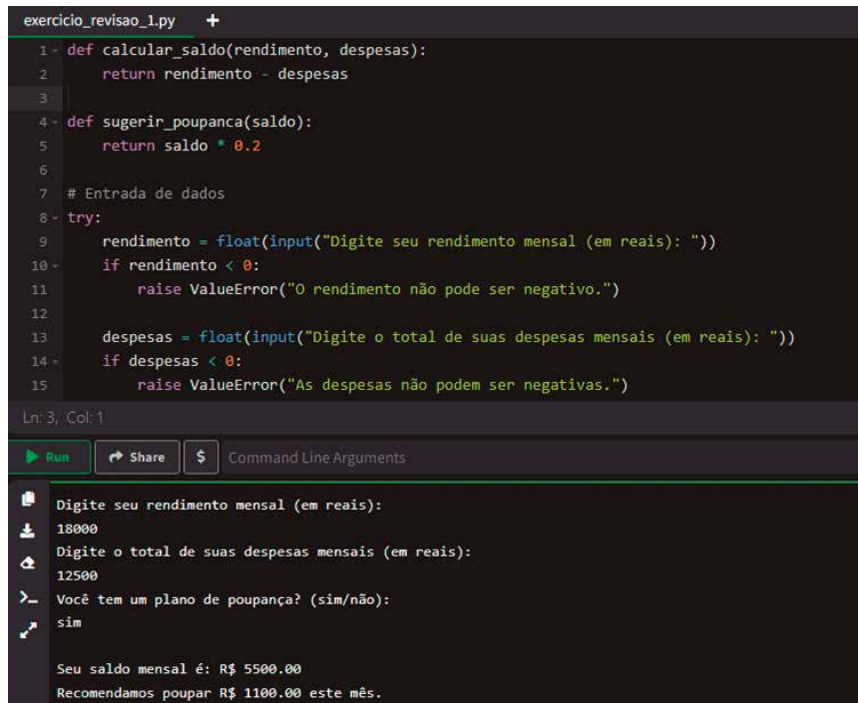
Finalmente, todo o bloco de entrada e processamento está dentro de um bloco `try-except`, que captura erros de entrada, como valores inválidos ou não numéricos. Ao capturar um erro, uma mensagem explicativa é exibida, detalhando o motivo do problema, e o programa não prossegue com os cálculos. Vamos ver isso com mais detalhes:

- A expressão `try` delimita um bloco cujas instruções correm o risco de falhar caso recebam valores inadequados. Pense em um cofre que guarda operações delicadas: enquanto o programa permanece dentro desse cofre, ele tenta realizar cálculos ou ler dados fornecidos pelo usuário.

Qualquer inconsistência identificada nesse perímetro é encaminhada para tratamento especial, evitando que a aplicação seja encerrada abruptamente.

- A instrução `raise` funciona como um alarme interno. Sempre que o programa detecta uma situação considerada inválida – por exemplo, rendimento negativo ou resposta fora das opções `sim` e `não` –, esse alarme é acionado através de `raise ValueError(...)`. Ao levantar um erro, o código informa explicitamente que encontrou um problema e repassa a responsabilidade de tratamento ao sistema de proteção que cerca o bloco.
- Ao final, surge `except`, que constitui o plano de contingência. Caso algum alarme seja acionado, o fluxo salta para a cláusula `except ValueError as e`. Dentro dela, o programador define uma resposta amigável: apresenta ao usuário uma mensagem esclarecendo qual foi a falha, a partir do conteúdo armazenado em `e`. Dessa forma, o programa continua vivo, orienta o usuário a corrigir a entrada e evita encerramentos inesperados.

Essa abordagem garante que o programa seja robusto e amigável ao usuário, lidando com diferentes cenários de entrada e oferecendo orientações claras com base na situação financeira apresentada. Na figura a seguir, apresentaremos um exemplo de execução desse código-fonte em um interpretador online.



```
exercicio_revisao_1.py +
1 def calcular_saldo(rendimento, despesas):
2     return rendimento - despesas
3
4 def sugerir_poupanca(saldo):
5     return saldo * 0.2
6
7 # Entrada de dados
8 try:
9     rendimento = float(input("Digite seu rendimento mensal (em reais): "))
10    if rendimento < 0:
11        raise ValueError("O rendimento não pode ser negativo.")
12
13    despesas = float(input("Digite o total de suas despesas mensais (em reais): "))
14    if despesas < 0:
15        raise ValueError("As despesas não podem ser negativas.")
16
17    saldo = calcular_saldo(rendimento, despesas)
18    poupanca = sugerir_poupanca(saldo)
19
20    print(f"Seu saldo mensal é: R$ {saldo:.2f}")
21    print(f"Recomendamos poupar R$ {poupanca:.2f} este mês.")
22 except ValueError as e:
23    print(e)
24
```

Ln: 3, Col: 1

Run Share Command Line Arguments

Digite seu rendimento mensal (em reais):
18000
Digite o total de suas despesas mensais (em reais):
12500
Você tem um plano de poupança? (sim/não):
sim
Seu saldo mensal é: R\$ 5500.00
Recomendamos poupar R\$ 1100.00 este mês.

Figura 3 – Exemplo da execução (parte inferior) do primeiro exercício no ambiente Online Python (<https://shre.ink/xefo>)

Exercício 2. Gerenciador de lista de compras

Imagine que você está desenvolvendo um programa para ajudar as pessoas a gerenciar suas listas de compras. Ele permitirá que o usuário adicione itens, remova itens e veja a lista completa antes de ir ao mercado. Crie um programa em Python que atenda aos seguintes requisitos.

- **Entrada de dados**

- Permitir que o usuário adicione itens à lista de compras.
- Possibilitar que o usuário remova itens da lista, caso estejam presentes.
- Oferecer a opção de listar todos os itens na lista.

- **Laços e iterações**

- Usar um loop para manter o programa em execução até que o usuário escolha sair.
- Utilizar laços para percorrer a lista e exibir os itens quando solicitado.

- **Operações com listas**

- Adicionar itens à lista usando o método `append`.
- Remover itens da lista usando o método `remove` ou verificações com `in`.
- Exibir todos os itens utilizando um laço `for`.

- **Tratamento de erros**

- Evitar que itens duplicados sejam adicionados à lista.
- Tratar situações em que o usuário tenta remover um item que não está na lista.

- **Instruções para o usuário**

1. Executar o programa.
2. Escolher uma opção do menu para adicionar, remover ou listar itens.
3. Testar cenários diferentes, tais como os indicados a seguir.
 - Adicionar itens já existentes.
 - Tentar remover itens que não estão na lista.
 - Listar itens quando a lista estiver vazia.
4. Explorar como o programa reage a entradas inválidas ou incorretas.

Essa atividade ajuda a praticar laços `for` e `while`, bem como operações com listas, entrada de dados e tratamento de erros de forma prática e contextualizada.

Em Python, os laços de repetição permitem que um conjunto de instruções seja executado de forma contínua, o que é fundamental para a automação de tarefas. O laço `for` é empregado para iterar sobre

seqüências, como listas ou `strings`, permitindo que se acesse cada elemento de forma ordenada. Já o laço `while` repete um bloco de código enquanto uma condição for verdadeira, sendo útil em situações em que não se sabe de antemão quantas vezes a repetição ocorrerá.

As operações com listas abrangem um conjunto de técnicas para manipular coleções de dados. Com elas, é possível criar, acessar, modificar, adicionar ou remover elementos; tornando as listas uma estrutura extremamente versátil para organizar informações em programas. Esse tipo de operação facilita o armazenamento e a gestão de conjuntos de dados dinâmicos, permitindo que os programas sejam mais flexíveis e adaptáveis a diferentes cenários.

Por fim, o tratamento de erros é uma prática essencial para o desenvolvimento de software robusto. Ele envolve a antecipação e a captura de exceções ou situações inesperadas que podem ocorrer durante a execução do programa. Com o tratamento adequado (usaremos `try/except`), é possível evitar que o programa seja interrompido abruptamente, fornecendo mensagens de erro esclarecedoras e, assim, melhorando a experiência do usuário e facilitando a manutenção do código.

A seguir, temos o código que soluciona o exercício.

```
def exibir_menu():
    print("\nMenu:")
    print("1. Adicionar item")
    print("2. Remover item")
    print("3. Listar itens")
    print("4. Sair")

def adicionar_item(lista):
    item = input("Digite o nome do item a ser adicionado: ").strip()
    if item in lista:
        print(f"O item '{item}' já está na lista.")
    else:
        lista.append(item)
        print(f"Item '{item}' adicionado com sucesso!")

def remover_item(lista):
    item = input("Digite o nome do item a ser removido: ").strip()
    if item in lista:
        lista.remove(item)
        print(f"Item '{item}' removido com sucesso!")
    else:
        print(f"O item '{item}' não está na lista.")

def listar_itens(lista):
    if lista:
        print("\nItens na lista de compras:")
        for i, item in enumerate(lista, start=1):
            print(f"{i}. {item}")
    else:
        print("A lista de compras está vazia.")

# Programa principal
lista_compras = []
```

```
while True:
    exibir_menu()
    try:
        opcao = int(input("Escolha uma opção: "))

        if opcao == 1:
            adicionar_item(lista_compras)
        elif opcao == 2:
            remover_item(lista_compras)
        elif opcao == 3:
            listar_itens(lista_compras)
        elif opcao == 4:
            print("Encerrando o programa. Boa sorte com suas compras!")
            break
        else:
            print("Opção inválida. Tente novamente.")
    except ValueError:
        print("Entrada inválida. Por favor, insira um número correspondente à opção.")
```

O código apresentado cria um programa que gerencia uma lista de compras, permitindo adicionar, remover e listar itens, com opções exibidas em um menu interativo. A estrutura do programa utiliza funções para organizar as operações principais, como exibir o menu, manipular os itens da lista e listar o conteúdo.

A função `exibir_menu` é responsável por apresentar ao usuário as opções disponíveis, exibindo uma mensagem com as quatro ações possíveis. Esse menu é mostrado repetidamente enquanto o programa está em execução e é chamado no início de cada iteração do laço principal (`while True: exibir_menu()`).

A função `adicionar_item` permite que o usuário insira um novo item na lista de compras. O nome do item é solicitado por meio da função `input`. O método `strip` é aplicado à entrada para remover espaços em branco no início e no final do texto digitado, o que evita problemas caso o usuário insira acidentalmente espaços extras. Após isso, o programa verifica se o item já está na lista. Se o item já existir, uma mensagem é exibida, informando que ele está duplicado. Caso contrário, o item é adicionado à lista com o método `append` e uma mensagem confirma a inclusão.

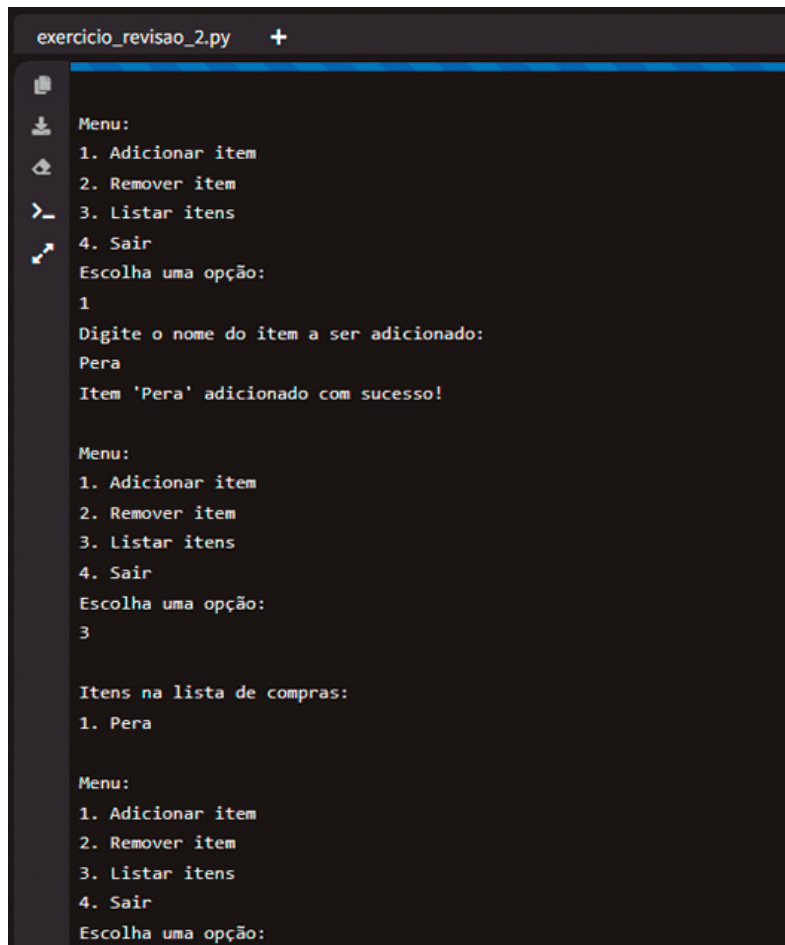
Na função `remover_item`, o programa solicita ao usuário o nome do item que deseja excluir. Assim como em `adicionar_item`, o método `strip` é aplicado para garantir que espaços adicionais não interfiram na identificação do item. Em seguida, verifica-se se o item está na lista. Caso esteja, ele é removido com o método `remove` e uma mensagem é exibida confirmando a remoção. Se o item não estiver presente, o programa informa que ele não foi encontrado na lista.

A função `listar_itens` exibe os itens atualmente armazenados na lista de compras. Antes de listar os itens, verifica-se se a lista está vazia. Caso não haja itens, uma mensagem é apresentada informando essa condição. Se houver itens, o programa utiliza um laço `for` para iterar sobre a lista. A função `enumerate` é empregada para numerar os itens automaticamente, começando a contagem pelo número 1. Isso resulta em uma exibição organizada, na qual cada item é mostrado com sua posição na lista.

No bloco principal do programa, uma lista vazia chamada `lista_compras` é inicializada para armazenar os itens que o usuário manipular. O programa entra em um laço `while` que permanece ativo até que o usuário escolha sair. A cada iteração, o menu é exibido por meio da função `exibir_menu` e o usuário deve selecionar uma opção digitando um número. A entrada é convertida para inteiro com a função `int` para que possa ser comparada com os números das opções.

O código utiliza uma estrutura condicional para tratar cada escolha. Se o usuário digitar 1, a função `adicionar_item` é chamada; para a opção 2, `remover_item` é executada; para a opção 3, `listar_itens` é acionada. Caso o usuário escolha a opção 4, uma mensagem de encerramento é exibida e o laço é interrompido com o comando `break`. Se for digitado um número que não corresponda a nenhuma opção, uma mensagem informa que a escolha é inválida.

O programa também inclui um bloco `try-except` para tratar erros relacionados à entrada do número da opção. Se o usuário inserir algo que não seja um número, como texto ou caracteres especiais, uma exceção `ValueError` será gerada. Nesse caso, o programa exibe uma mensagem de erro e solicita uma nova tentativa. Na figura a seguir, apresentaremos um exemplo de execução desse código-fonte.



```
exercicio_revisao_2.py +
Menu:
1. Adicionar item
2. Remover item
3. Listar itens
4. Sair
Escolha uma opção:
1
Digite o nome do item a ser adicionado:
Pera
Item 'Pera' adicionado com sucesso!

Menu:
1. Adicionar item
2. Remover item
3. Listar itens
4. Sair
Escolha uma opção:
3

Itens na lista de compras:
1. Pera

Menu:
1. Adicionar item
2. Remover item
3. Listar itens
4. Sair
Escolha uma opção:
```

Figura 4 – Exemplo da execução do segundo exercício no ambiente Online Python (<https://shre.ink/xeme>)

Exercício 3. Gerenciador de tarefas diárias

Imagine que você está criando um programa para ajudar as pessoas a organizarem suas tarefas diárias. Ele deve permitir que o usuário adicione tarefas, marque tarefas como concluídas e visualize tarefas pendentes ou concluídas. Crie um programa em Python que atenda aos seguintes requisitos listados a seguir.

- **Entrada de dados**

- Permitir que o usuário adicione tarefas à lista de tarefas.
- Possibilitar que o usuário marque tarefas como concluídas.
- Oferecer a opção de visualizar todas as tarefas pendentes ou concluídas.

- **Controle de fluxo aplicado**

- Usar estruturas de controle (`if`, `else`, `elif`) para gerenciar as escolhas do usuário no menu principal.
- Diferenciar entre tarefas pendentes e concluídas com base no estado delas.

- **Operações com listas**

- Usar uma lista para armazenar as tarefas e o estado de cada uma (pendente ou concluída).
- Realizar operações como adição, remoção e modificação de itens na lista.

- **Tratamento de erros**

- Evitar que tarefas duplicadas sejam adicionadas.
- Lidar com tentativas de marcar como concluídas tarefas que não existem.

- **Instruções para o usuário**

1. Executar o programa.
2. Escolher uma opção do menu para adicionar, marcar como concluída ou listar tarefas.
3. Testar diferentes cenários:
 - Adicionar tarefas com descrições iguais.

- Marcar como concluída uma tarefa inexistente.
 - Listar tarefas quando todas forem concluídas ou quando não houver nenhuma tarefa.
4. Observar como o programa utiliza controle de fluxo para responder às entradas do usuário e gerencia as operações com listas.

Essa atividade explora o controle de fluxo aplicado a situações práticas, bem como operações com listas e organização de dados. Ela também desafia o aluno a pensar em condições e estados diferentes em um sistema simples. A seguir, apresentamos o código-fonte que soluciona o exercício.

```
def exibir_menu():
    print("\nMenu:")
    print("1. Adicionar tarefa")
    print("2. Marcar tarefa como concluída")
    print("3. Listar tarefas pendentes")
    print("4. Listar tarefas concluídas")
    print("5. Sair")

def adicionar_tarefa(lista_tarefas):
    tarefa = input("Digite a descrição da tarefa: ").strip()
    if any(tarefa == item['descricao'] for item in lista_tarefas):
        print(f"A tarefa '{tarefa}' já está na lista.")
    else:
        lista_tarefas.append({"descricao": tarefa, "concluida": False})
        print(f"Tarefa '{tarefa}' adicionada com sucesso!")

def marcar_concluida(lista_tarefas):
    tarefa = input("Digite a descrição da tarefa a ser marcada como concluída: ").strip()
    for item in lista_tarefas:
        if item["descricao"] == tarefa:
            if item["concluida"]:
                print(f"A tarefa '{tarefa}' já está marcada como concluída.")
            else:
                item["concluida"] = True
                print(f"Tarefa '{tarefa}' marcada como concluída com sucesso!")
    return
    print(f"Tarefa '{tarefa}' não encontrada na lista.")

def listar_tarefas(lista_tarefas, concluida):
    status = "concluidas" if concluida else "pendentes"
    tarefas = [item["descricao"] for item in lista_tarefas if item["concluida"]
    == concluida]

    if tarefas:
        print(f"\nTarefas {status}:")
        for i, tarefa in enumerate(tarefas, start=1):
            print(f"{i}. {tarefa}")
    else:
        print(f"Não há tarefas {status} no momento.")

# Programa principal
lista_tarefas = []
```

```
while True:
    exibir_menu()
    try:
        opcao = int(input("Escolha uma opção: "))

        if opcao == 1:
            adicionar_tarefa(lista_tarefas)
        elif opcao == 2:
            marcar_concluida(lista_tarefas)
        elif opcao == 3:
            listar_tarefas(lista_tarefas, concluida=False)
        elif opcao == 4:
            listar_tarefas(lista_tarefas, concluida=True)
        elif opcao == 5:
            print("Encerrando o programa. Boa sorte com suas tarefas!")
            break
        else:
            print("Opção inválida. Tente novamente.")
    except ValueError:
        print("Entrada inválida. Por favor, insira um número correspondente à opção.")
```

O código apresentado é organizado em diversas funções que, juntas, formam uma interface de linha de comando interativa. A função que exibe o menu apresenta as opções disponíveis, imprimindo uma lista numerada com comandos para adicionar uma nova tarefa, marcar uma tarefa como concluída, listar as tarefas pendentes, listar as tarefas concluídas e sair do programa. Cada uma dessas ações é implementada em funções separadas para tornar o código modular e mais fácil de entender.

Na função responsável por adicionar uma tarefa, o programa solicita ao usuário a descrição da tarefa por meio da função `input`, utilizando o método `strip` para remover qualquer espaço em branco extra no início ou no final da entrada. Esse cuidado evita que tarefas sejam consideradas diferentes apenas por causa de espaços indesejados. Em seguida, a função utiliza a expressão `any` com uma expressão geradora (generator expression) para verificar se a tarefa informada já existe na lista, comparando a descrição digitada com a descrição de cada tarefa armazenada, que está representada por um dicionário. Se a tarefa já existir, uma mensagem informando esse fato é exibida; caso contrário, a tarefa é adicionada à lista como um dicionário contendo a descrição e uma chave que indica se ela está concluída ou não, iniciando com o valor `False`.



Observação

O conceito generator expression é um recurso da Python que permite criar coleções de maneira concisa e eficiente. Seu uso evita a necessidade de laços explícitos (`for`) e torna o código mais legível. Em vez de criar e armazenar a lista inteira na memória, ele gera os elementos sob demanda, economizando espaço. Por exemplo, ao invés de escrever um laço `for` tradicional para criar uma lista de quadrados dos números de 1 a 10, podemos usar o generator expression:

```
quadrados_gen = (x**2 for x in range(1, 11))
print(quadrados_gen) # Retorna um objeto gerador
print(next(quadrados_gen)) # Obtém o primeiro valor gerado.
Ou seja, 1 elevado ao quadrado que é 1.
print(next(quadrados_gen)) # Obtém o segundo valor gerado.
Ou seja, 2 elevado ao quadrado que é 4.
```

Se for necessário iterar por todos os valores, pode-se usar um for:

```
for num in quadrados_gen:
    print(num)
```

A vantagem da expressão geradora é que os valores não são armazenados em uma lista, mas sim calculados conforme necessário. Essa característica torna o método muito mais eficiente em termos de uso de memória, especialmente quando se trabalha com conjuntos de dados grandes.

Para marcar uma tarefa como concluída, a função correspondente solicita ao usuário a descrição da tarefa que deseja alterar e, novamente, utiliza o método `strip` para garantir que a comparação não seja afetada por espaços em branco. A função percorre a lista de tarefas com um laço `for`, que avalia cada dicionário. Se encontrar um dicionário cuja descrição corresponda à entrada do usuário, o programa verifica se a tarefa está marcada como concluída. Se já estiver, uma mensagem apropriada é exibida; caso contrário, a chave que indica a conclusão da tarefa é alterada para `True` e uma mensagem confirmando a operação é impressa. Se nenhuma tarefa com a descrição fornecida for encontrada, o usuário é informado de que a tarefa não está na lista.

A função que lista as tarefas recebe um parâmetro que indica se devem ser listadas as tarefas concluídas ou as tarefas pendentes. Nela, é utilizada uma compreensão de lista (list comprehension) para criar uma lista contendo apenas as descrições das tarefas filtradas de acordo com seu estado de conclusão. Se houver tarefas que atendam ao critério, a função utiliza um laço `for` com a função `enumerate` para percorrer essa lista, atribuindo um número sequencial a cada tarefa e facilitando a leitura da lista pelo usuário, já que a contagem começa em 1. Se não houver tarefas no estado requisitado, uma mensagem é exibida informando que não há tarefas naquele momento.



Observação

Os conceitos de list comprehension e generator expression são recursos do Python que evitam a necessidade de laços explícitos (for) e tornam o código mais legível. No entanto, há diferenças fundamentais entre eles, principalmente no modo como armazenam e processam os dados. A list comprehension é uma maneira compacta de criar listas a partir de uma sequência de elementos, aplicando transformações e filtrações diretamente na declaração da lista. Sua principal característica é que ela retorna

uma nova lista contendo todos os elementos gerados pela expressão interna. Como o resultado é armazenado diretamente na memória, esse método pode ser menos eficiente quando a quantidade de elementos for muito grande.

Por exemplo, ao invés de escrever um laço for tradicional para criar uma lista de quadrados dos números de 1 a 10, pode-se usar uma compreensão de lista:

```
quadrados = [x**2 for x in range(1, 11)]  
print(quadrados)
```

Esse código percorre os números de 1 a 10 e eleva cada um ao quadrado, armazenando todos os valores na lista quadrados. A sintaxe segue a estrutura:

```
[nova_expressão for item in iterável if condição]
```

Se houver uma condição, somente os elementos que a satisfazem serão incluídos na nova lista. O exemplo a seguir gera uma lista contendo apenas os números pares de 1 a 10.

```
pares = [x for x in range(1, 11) if x % 2 == 0]  
print(pares)
```

No bloco principal, o programa inicializa uma lista vazia para armazenar as tarefas e entra em um laço infinito que apresenta o menu a cada iteração. O usuário escolhe uma opção digitando um número, que é convertido para inteiro. Esse processo de conversão é protegido por um bloco try-except, de modo que se o usuário digitar algo que não possa ser convertido, o programa captura a exceção ValueError e exibe uma mensagem de erro, solicitando uma nova tentativa. Dependendo do número digitado, o programa chama a função correspondente: adiciona uma tarefa, marca uma tarefa como concluída, lista as tarefas pendentes ou concluídas, ou encerra o programa caso a opção seja sair. Se o número não corresponder a nenhuma opção válida, uma mensagem informa que a opção é inválida. Na figura 5, apresentaremos um exemplo da execução do programa completo.

```
Menu:
1. Adicionar tarefa
2. Marcar tarefa como concluída
3. Listar tarefas pendentes
4. Listar tarefas concluídas
5. Sair
Escolha uma opção:
1
Digite a descrição da tarefa:
Estudar Python
Tarefa 'Estudar Python' adicionada com sucesso!

Menu:
1. Adicionar tarefa
2. Marcar tarefa como concluída
3. Listar tarefas pendentes
4. Listar tarefas concluídas
5. Sair
Escolha uma opção:
3

Tarefas pendentes:
1. Estudar Python
```

Figura 5 – Exemplo da execução do terceiro exercício no ambiente Online Python (<https://shre.ink/xe3q>)

Exercício 4. Agenda de contatos

Imagine que você está desenvolvendo um programa para armazenar contatos telefônicos. Ele deve permitir que o usuário adicione, remova e visualize contatos, além de salvá-los e carregá-los de um arquivo para garantir que as informações sejam preservadas entre as execuções. Crie um programa em Python que atenda aos seguintes requisitos.

- **Manipulação de dicionários**
 - Armazenar os contatos em um dicionário, no qual a chave será o nome da pessoa e o valor será outro dicionário contendo telefone e e-mail.
- **Manipulação de arquivos**
 - Salvar os contatos em um arquivo JSON para garantir persistência de dados.
 - Carregar os contatos do arquivo JSON ao iniciar o programa.
- **Funcionalidades**
 - Adicionar um novo contato (nome, telefone e e-mail).

- Remover um contato existente.
- Listar todos os contatos armazenados.
- Salvar automaticamente as alterações no arquivo.

- **Tratamento de erros**

- Impedir a duplicação de contatos.
- Lidar com a tentativa de remoção de um contato inexistente.
- Tratar problemas de leitura e escrita no arquivo.

- **Instruções para o usuário**

1. Executar o programa.
2. Escolher uma opção do menu para adicionar, remover ou listar contatos.
3. O programa salva automaticamente os contatos no arquivo `contatos.json`, permitindo que os dados persistam após o fechamento.
4. Testar cenários diferentes:
 - Adicionar um contato já existente.
 - Remover um contato que não está na lista.
 - Fechar e reabrir o programa para verificar a persistência dos dados.

Esse exercício permite ao aluno explorar a manipulação de dicionários para armazenar informações estruturadas, além de praticar a leitura e a escrita de arquivos JSON, garantindo persistência de dados entre diferentes execuções.

JSON (JavaScript Object Notation) é um formato de armazenamento e troca de dados que organiza informações de maneira estruturada e de fácil leitura, tanto para humanos quanto para computadores. Ele utiliza uma estrutura baseada em pares chave-valor, semelhante a dicionários em Python ou objetos em JavaScript, e permite representar dados complexos, como listas e hierarquias aninhadas. Esse formato é amplamente utilizado em comunicação entre sistemas, especialmente em APIs (Application Programming Interface), pois é leve e compatível com diversas linguagens de programação.



Observação

Uma API (Interface de Programação de Aplicações) é um conjunto de regras e definições que permite que diferentes sistemas e programas se comuniquem entre si. Imagine uma API como um garçom em um restaurante: você faz um pedido (solicitação de dados ou serviço), o garçom (API) leva esse pedido até a cozinha (o sistema que processa a informação) e depois traz a resposta até você. No mundo da tecnologia, as APIs são usadas para conectar aplicativos, sites e dispositivos a serviços online, permitindo, por exemplo, que um aplicativo de clima acesse dados meteorológicos de um servidor ou que um site faça login utilizando uma conta do Google. Esse mecanismo facilita a integração entre diferentes plataformas sem que o usuário precise saber como cada sistema funciona internamente.

Um arquivo JSON deve seguir uma sintaxe específica, na qual chaves e strings são delimitadas por aspas duplas, números são escritos sem aspas e valores booleanos são representados como `true` ou `false`.

A seguir, um exemplo de arquivo JSON representando uma lista de tarefas:

```
{
  "tarefas": [
    {
      "descricao": "Comprar pão",
      "prioridade": "alta",
      "concluida": false
    },
    {
      "descricao": "Estudar Python",
      "prioridade": "média",
      "concluida": true
    },
    {
      "descricao": "Fazer exercícios",
      "prioridade": "baixa",
      "concluida": false
    }
  ]
}
```

No exemplo anterior, há um objeto principal contendo uma lista de tarefas. Cada tarefa é representada por um dicionário com três informações: a descrição da atividade, o nível de prioridade e se a tarefa foi concluída ou não. Esse formato facilita a organização e a transmissão de dados entre diferentes sistemas.



Saiba mais

O livro a seguir aborda o trabalho com arquivos CSV e dados JSON, ensinando como ler e escrever dados nesses formatos utilizando Python.

SWEIGART, A. *Automatize tarefas maçantes com Python*: programação prática para verdadeiros iniciantes. São Paulo: Novatec, 2015.

Outra referência importante é a obra a seguir, que oferece diversas receitas práticas para programadores Python, incluindo técnicas para manipulação de dados JSON, leitura e escrita de arquivos JSON e integração com APIs que utilizam esse formato.

BEAZLEY, D.; JONES, B. K. *Python cookbook*: recipes for mastering Python 3. 3. ed. [s.l.]: O'Reilly Media, 2013.

Por fim, vale a pena conferir o livro de Mark Lutz. Nesta obra abrangente sobre Python, há seções dedicadas ao trabalho com dados JSON, explicando como utilizar o módulo JSON da biblioteca padrão para serializar e desserializar dados, além de exemplos práticos de uso.

LUTZ, M. *Learning Python*. 5. ed. Sebastopol: O'Reilly Media, 2013.

Nos três primeiros exercícios da programação Python deste livro-texto usamos uma ferramenta online para digitação do código-fonte e execução no interpretador Python. Neste exercício, faremos diferente. Usaremos uma plataforma um pouco mais robusta, que exige a instalação do Visual Studio Code (VS Code). Ao trabalhar com leitura e escrita de arquivos em Python, o VS Code é uma opção superior aos compiladores online porque permite acesso direto ao sistema de arquivos do computador.

Ele é um editor de código gratuito e altamente personalizável, desenvolvido pela Microsoft, utilizado para programação em diversas linguagens, incluindo Python. O VS Code oferece recursos como realce de sintaxe, depuração integrada, suporte a extensões e integração com sistemas de controle de versão, tornando o desenvolvimento mais produtivo e eficiente.

No VS Code, como o código é executado no próprio computador do usuário, é possível ler e gravar arquivos sem restrições, utilizando comandos como `open()`, `read()`, `write()` e `close()`, garantindo que os dados sejam armazenados de forma permanente no diretório desejado. Além disso, o VS Code oferece suporte à depuração, facilitando a identificação de erros ao lidar com arquivos, e possibilita a visualização direta dos arquivos criados ou modificados dentro do próprio editor. Para quem está aprendendo a manipular arquivos em Python, essa flexibilidade é essencial para testar, entender e corrigir o código de forma eficiente.

No entanto, para programar em Python no Windows, é necessário instalar, além do VS Code, o interpretador Python, pois o editor, por si só, não consegue executar código Python sem essa ferramenta. O interpretador Python pode ser obtido no site oficial (<https://shre.ink/xehV>) e sua instalação garante que comandos e scripts em Python sejam executados corretamente no sistema. Após instalar o interpretador, recomenda-se adicionar a extensão oficial do Python no VS Code, permitindo funcionalidades avançadas, como autocompletar código, realce de erros e depuração interativa.



Observação

Além do VS Code, existem outras opções populares para desenvolvimento em Python, cada uma com características específicas. O PyCharm, desenvolvido pela JetBrains, é um ambiente de desenvolvimento integrado (IDE) focado em Python, oferecendo recursos avançados, como depuração poderosa, análise de código e integração com frameworks populares, sendo ideal para projetos mais complexos. Outra alternativa é o Thonny, uma IDE leve e intuitiva voltada para iniciantes, que já vem com um interpretador Python embutido, simplificando a configuração inicial. O IDLE, que acompanha a instalação oficial do Python, é uma opção básica, mas eficiente para pequenos scripts e aprendizado inicial. Também há o Jupyter Notebook, amplamente utilizado em ciência de dados e aprendizado de máquina, permitindo a execução interativa de código junto com explicações em texto e gráficos. Cada ferramenta possui vantagens específicas, dependendo das necessidades do programador e do tipo de projeto.

Em Python, os dicionários são estruturas fundamentais que permitem armazenar dados de forma organizada, utilizando pares de chave-valor. Essa estrutura é especialmente útil quando se deseja associar informações a identificadores únicos, como nomes ou IDs, possibilitando acesso rápido e intuitivo aos dados.

A persistência dos dados é alcançada por meio do uso do módulo `json`, que integra a conversão entre dicionários e o formato JSON.



Lembrete

Como vimos, esse formato é amplamente utilizado para armazenar e transmitir informações, pois é leve e fácil de ler tanto para humanos quanto para máquinas.

Em Python, funções como `json.dump()` e `json.load()` simplificam esse processo, permitindo que o conteúdo de um dicionário seja salvo em um arquivo e, posteriormente, recuperado sem a perda da estrutura original. Essa capacidade de ler e escrever arquivos JSON garante que as informações

manipuladas pelo programa sejam mantidas entre diferentes execuções, o que é crucial para aplicações que necessitam de armazenamento persistente.

A seguir, apresentaremos o código-fonte que soluciona nosso exercício da agenda de contatos.

```
import json
import os

ARQUIVO_CONTATOS = "contatos.json"
def carregar_contatos():
    if os.path.exists(ARQUIVO_CONTATOS):
        try:
            with open(ARQUIVO_CONTATOS, "r", encoding="utf-8")
as arquivo:
                return json.load(arquivo)
        except json.JSONDecodeError:
            print("Erro ao carregar os contatos. O arquivo pode estar corrompido.")
            return {}
    return {}

def salvar_contatos(contatos):
    with open(ARQUIVO_CONTATOS, "w", encoding="utf-8") as arquivo:
        json.dump(contatos, arquivo, indent=4, ensure_ascii=False)

def adicionar_contato(contatos):
    nome = input("Digite o nome do contato: ").strip()
    if nome in contatos:
        print(f"O contato '{nome}' já existe.")
        return

    telefone = input("Digite o telefone: ").strip()
    email = input("Digite o e-mail: ").strip()

    contatos[nome] = {"telefone": telefone, "email": email}
    salvar_contatos(contatos)
    print(f"Contato '{nome}' adicionado com sucesso!")

def remover_contato(contatos):
    nome = input("Digite o nome do contato a ser removido: ").strip()
    if nome in contatos:
        del contatos[nome]
        salvar_contatos(contatos)
        print(f"Contato '{nome}' removido com sucesso!")
    else:
        print(f"O contato '{nome}' não foi encontrado.")

def listar_contatos(contatos):
    if contatos:
        print("\nLista de Contatos:")
        for nome, dados in contatos.items():
            print(f"Nome: {nome}, Telefone: {dados['telefone']}, E-mail: {dados['email']}")
    else:
        print("Nenhum contato cadastrado.")
```

```
def exibir_menu():
    print("\nMenu:")
    print("1. Adicionar contato")
    print("2. Remover contato")
    print("3. Listar contatos")
    print("4. Sair")

# Programa principal
contatos = carregar_contatos()

while True:
    exibir_menu()
    try:
        opcao = int(input("Escolha uma opção: "))

        if opcao == 1:
            adicionar_contato(contatos)
        elif opcao == 2:
            remover_contato(contatos)
        elif opcao == 3:
            listar_contatos(contatos)
        elif opcao == 4:
            print("Encerrando o programa. Seus contatos foram salvos.")
            break
        else:
            print("Opção inválida. Tente novamente.")
    except ValueError:
        print("Entrada inválida. Por favor, insira um número correspondente à opção.")
```

O código apresentado é um exemplo de sistema simples de gerenciamento de contatos que utiliza a persistência de dados por meio de um arquivo JSON.

Inicialmente, são importados os módulos `json` e `os`, sendo que o primeiro é usado para converter objetos Python em uma representação em texto (e vice-versa) e o segundo para interagir com o sistema operacional, como verificar a existência de um arquivo.

Uma constante define o nome do arquivo no qual os contatos serão salvos, permitindo que ele seja referenciado de forma consistente em todas as funções.

A função responsável por carregar os contatos verifica se o arquivo existe utilizando uma função do módulo `os` (`os.path.exists`); caso exista, tenta abrir e carregar o conteúdo com `json.load`, transformando o conteúdo do arquivo de volta em um dicionário Python. Se ocorrer algum problema na decodificação do JSON – por exemplo, se o arquivo estiver corrompido –, uma exceção específica é capturada e o programa informa o usuário, retornando um dicionário vazio para evitar a interrupção inesperada do programa.

Por sua vez, a função de salvar os contatos recebe o dicionário atualizado e utiliza `json.dump` para escrever os dados de volta no arquivo, formatando a saída com indentação para facilitar a leitura e garantindo que os caracteres especiais sejam preservados, já que o parâmetro `ensure_ascii` está definido como `False`.

As funções de adicionar e remover contatos trabalham com o dicionário que armazena os dados. Ao adicionar um contato, o programa solicita ao usuário nome, telefone e e-mail, garantindo que não haja duplicidade de nomes, e em seguida atualiza o dicionário e salva as alterações.

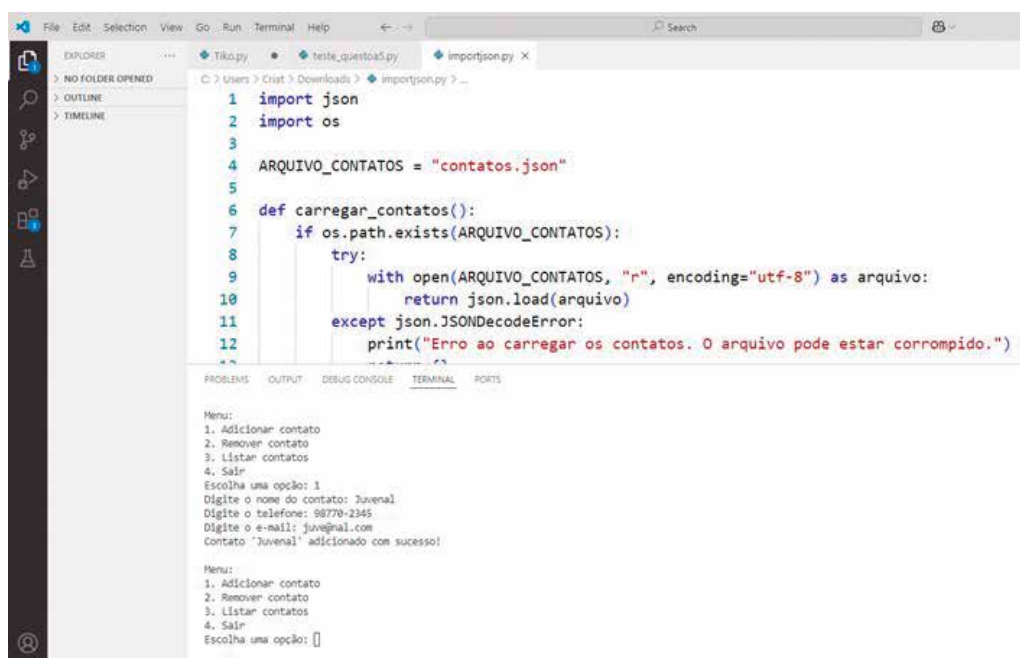
A remoção de um contato é feita verificando se o nome informado existe no dicionário; se existir, o contato é removido, e o novo estado dos dados é escrito no arquivo, caso contrário, uma mensagem de erro é exibida. A função para listar os contatos percorre o dicionário e imprime as informações de cada contato de forma organizada, mas exibe uma mensagem apropriada se não houver registro cadastrado.

O fluxo principal do programa inicia carregando os contatos já salvos e, em seguida, entra em um loop que apresenta repetidamente um menu de opções para o usuário. Esse menu permite escolher entre adicionar, remover, listar contatos ou encerrar o programa.

A entrada do usuário é convertida para inteiro e, utilizando estruturas condicionais, o programa direciona a execução para a função correspondente à opção escolhida.

O uso de um bloco `try/except` assegura que, se o usuário digitar um valor que não possa ser convertido em número, uma mensagem de erro seja exibida sem que o programa seja encerrado abruptamente. Assim, o código demonstra uma abordagem modular e resiliente, na qual cada função tem uma responsabilidade específica e a persistência dos dados é garantida por meio de operações de leitura e escrita em um arquivo JSON, facilitando a manutenção e a compreensão do sistema como um todo.

Na figura a seguir, apresentaremos um exemplo da execução do programa completo, no ambiente VS Code. Note que o código-fonte foi digitado na parte superior da figura e executado na parte inferior.



```
1 import json
2 import os
3
4 ARQUIVO_CONTATOS = "contatos.json"
5
6 def carregar_contatos():
7     if os.path.exists(ARQUIVO_CONTATOS):
8         try:
9             with open(ARQUIVO_CONTATOS, "r", encoding="utf-8") as arquivo:
10                 return json.load(arquivo)
11         except json.JSONDecodeError:
12             print("Erro ao carregar os contatos. O arquivo pode estar corrompido.")
```

Menu:

1. Adicionar contato
2. Remover contato
3. Listar contatos
4. Sair

Escolha uma opção: 1
Digite o nome do contato: Juvenal
Digite o telefone: 98770-2345
Digite o e-mail: juve@mail.com
Contato "Juvenal" adicionado com sucesso!

Menu:

1. Adicionar contato
2. Remover contato
3. Listar contatos
4. Sair

Escolha uma opção:

Figura 6 – Exemplo da execução do quarto exercício no ambiente VS Code

Exercício 5. Sistema de biblioteca

Imagine que você está desenvolvendo um sistema para gerenciar uma biblioteca. O sistema deve permitir que o usuário cadastre, empreste e devolva livros; além de visualizar a lista de livros disponíveis e emprestados. Crie um programa em Python que atenda aos seguintes requisitos.

- **Aplicação prática de POO (Programação Orientada a Objetos)**

- Criar uma classe `Livro` para representar os livros da biblioteca.
- Criar uma classe `Biblioteca` para gerenciar a coleção de livros e operações.

- **Funcionalidades**

- Adicionar novos livros à biblioteca.
- Listar todos os livros disponíveis.
- Empréstimo de um livro (marcando-o como emprestado).
- Devolver um livro (tornando-o disponível novamente).
- Listar os livros emprestados.

- **Uso de métodos e encapsulamento**

- Criar métodos dentro da classe `Biblioteca` para encapsular a lógica de manipulação de livros.
- Utilizar atributos privados ou protegidos para evitar manipulação direta indevida.

- **Tratamento de erros**

- Evitar que um livro já emprestado seja emprestado novamente.
- Evitar a devolução de um livro que não está emprestado.
- Garantir que o livro exista antes de realizar operações sobre ele.

- **Instruções para o usuário**

1. Executar o programa.
2. Escolher uma opção do menu para adicionar, listar, emprestar ou devolver livros.

3. Testar diferentes cenários:

- Adicionar um livro já existente e verificar se há alguma restrição.
- Empréstimo de um livro que já está emprestado.
- Devolver um livro que não está emprestado ou que não existe.
- Listar livros para verificar se o status muda corretamente.

Esse exercício revisa os conceitos básicos da POO, incluindo a criação de classes, atributos, métodos e encapsulamento, aplicados a um cenário realista e útil.

Ao definir uma classe, o programador cria um molde para objetos que possuem tanto características – representadas pelos atributos – quanto comportamentos – expressos pelos métodos. Essa abordagem permite que cada objeto seja uma instância com seus próprios dados, facilitando a organização e o reuso do código.

A criação de classes em Python é feita de maneira bastante direta, utilizando a palavra-chave `def` para métodos e a convenção de usar o `self` como referência ao objeto atual. Os atributos podem ser definidos tanto no momento da criação do objeto, através do método `__init__`, quanto adicionados posteriormente, sempre que necessário. Essa flexibilidade é característica da linguagem, tornando a manipulação de dados internos dos objetos simples e intuitiva.

O encapsulamento é um princípio fundamental da POO, que, em Python, é incentivado por meio de convenções. Embora a linguagem não imponha restrições rígidas de acesso, a utilização de underscores antes do nome de um atributo ou método indica que ele é destinado ao uso interno da classe, sugerindo que não deve ser acessado diretamente de fora. Essa prática ajuda a manter uma interface limpa e clara, na qual a implementação interna pode ser alterada sem impactar o restante do programa.

A seguir, apresentaremos o código-fonte que soluciona o exercício.

```
class Livro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor
        self.emprestado = False

    def __str__(self):
        status = "Emprestado" if self.emprestado else "Disponível"
        return f"{self.titulo} - {self.autor} ({status})"

class Biblioteca:
    def __init__(self):
        self.livros = []

    def adicionar_livro(self, titulo, autor):
```

```
self.livros.append(Livro(titulo, autor))
print(f"Livro '{titulo}' adicionado à biblioteca.")

def listar_livros_disponiveis(self):
    disponiveis = [livro for livro in self.livros if not livro.emprestado]
    if disponiveis:
        print("\nLivros disponíveis:")
        for livro in disponiveis:
            print(f"- {livro}")
    else:
        print("Nenhum livro disponível no momento.")

def listar_livros_emprestados(self):
    emprestados = [livro for livro in self.livros if livro.emprestado]
    if emprestados:
        print("\nLivros emprestados:")
        for livro in emprestados:
            print(f"- {livro}")
    else:
        print("Nenhum livro emprestado no momento.")

def emprestar_livro(self, titulo):
    for livro in self.livros:
        if livro.titulo.lower() == titulo.lower():
            if not livro.emprestado:
                livro.emprestado = True
                print(f"Livro '{titulo}' emprestado com sucesso.")
            else:
                print(f"O livro '{titulo}' já está emprestado.")
            return
    print(f"O livro '{titulo}' não foi encontrado na biblioteca.")

def devolver_livro(self, titulo):
    for livro in self.livros:
        if livro.titulo.lower() == titulo.lower():
            if livro.emprestado:
                livro.emprestado = False
                print(f"Livro '{titulo}' devolvido com sucesso.")
            else:
                print(f"O livro '{titulo}' não estava emprestado.")
            return
    print(f"O livro '{titulo}' não foi encontrado na biblioteca.")

def exibir_menu():
    print("\nMenu:")
    print("1. Adicionar livro")
    print("2. Listar livros disponíveis")
    print("3. Listar livros emprestados")
    print("4. Emprestar livro")
    print("5. Devolver livro")
    print("6. Sair")

# Programa principal
biblioteca = Biblioteca()
```

```
while True:
    exibir_menu()
    try:
        opcao = int(input("Escolha uma opção: "))

        if opcao == 1:
            titulo = input("Digite o título do livro: ").strip()
            autor = input("Digite o autor do livro: ").strip()
            biblioteca.adicionar_livro(titulo, autor)
        elif opcao == 2:
            biblioteca.listar_livros_disponiveis()
        elif opcao == 3:
            biblioteca.listar_livros_emprestados()
        elif opcao == 4:
            titulo = input("Digite o título do livro a ser emprestado: ").strip()
            biblioteca.emprestar_livro(titulo)
        elif opcao == 5:
            titulo = input("Digite o título do livro a ser devolvido: ").strip()
            biblioteca.devolver_livro(titulo)
        elif opcao == 6:
            print("Encerrando o programa. Volte sempre à biblioteca!")
            break
        else:
            print("Opção inválida. Tente novamente.")
    except ValueError:
        print("Entrada inválida. Por favor, insira um número correspondente à opção.")
```

O código apresentado implementa um sistema simples de gerenciamento de livros para uma biblioteca, permitindo adicionar novos títulos, listar livros disponíveis e emprestados, realizar empréstimos e devoluções. O programa é estruturado utilizando classes, organizando os dados e funcionalidades de forma eficiente e intuitiva.

A classe `Livro` representa um livro dentro da biblioteca, contendo três atributos principais: `titulo`, `autor` e `emprestado`. Os dois primeiros armazenam o nome e o autor do livro, enquanto `emprestado` indica se ele está ou não emprestado, assumindo inicialmente o valor `False`, ou seja, o livro começa disponível.

O método especial `__init__` é responsável por inicializar esses atributos quando um objeto `Livro` é criado. Além disso, há a definição do método `__str__`, que retorna uma representação textual do livro, incluindo seu título, autor e status (disponível ou emprestado). Esse método facilita a exibição das informações do livro quando ele é impresso.

A classe `Biblioteca` gerencia uma coleção de livros. Seu construtor `__init__` inicializa um atributo `livros`, que é uma lista vazia na qual os objetos `Livro` serão armazenados. Os métodos dessa classe permitem manipular a lista de livros, implementando as operações essenciais do sistema.

O método `adicionar_livro` recebe um título e um autor como argumentos, cria um objeto da classe `Livro` e o adiciona à lista de livros. Após essa adição, uma mensagem confirma que o livro foi incluído com sucesso.

O método `listar_livros_disponiveis` verifica quais livros na biblioteca ainda não foram emprestados. Ele utiliza uma compreensão de lista para criar uma lista apenas com os livros cujo atributo `emprestado` é `False`. Caso existam livros disponíveis, eles são exibidos um por um, utilizando um laço `for`. Como a classe `Livro` possui o método `__str__`, a impressão dos objetos nessa lista já inclui as informações formatadas. Se não houver livros disponíveis, uma mensagem informa essa condição ao usuário.

O método `listar_livros_emprestados` funciona de maneira semelhante ao método anterior, porém filtrando apenas os livros que possuem o atributo `emprestado` definido como `True`. Se houver livros emprestados, eles são exibidos da mesma forma, e se não houver nenhum, uma mensagem apropriada é exibida.

O método `emprestar_livro` permite que um usuário solicite um livro emprestado. O programa percorre a lista de livros e compara o título informado com o título de cada livro armazenado. A comparação ignora maiúsculas e minúsculas ao converter ambos os valores para letras minúsculas com `lower()`. Se o livro for encontrado e estiver disponível, seu atributo `emprestado` é alterado para `True` e uma mensagem confirma que o empréstimo foi realizado. Caso o livro já esteja emprestado, o usuário é informado de que ele não está disponível no momento. Se nenhum livro correspondente for encontrado na biblioteca, o programa informa que o título não foi localizado.

O método `devolver_livro` funciona de maneira análoga ao de empréstimo. Ele percorre a lista de livros em busca do título informado. Se o livro for encontrado e estiver emprestado, seu atributo `emprestado` é redefinido para `False`, indicando que ele voltou a estar disponível. Caso o livro seja encontrado, mas não esteja emprestado, o usuário é informado de que a devolução não era necessária. Se o título não for localizado, o programa informa que o livro não foi encontrado na biblioteca.

A função `exibir_menu` apresenta ao usuário um menu com as opções disponíveis no sistema. Essa função é chamada repetidamente dentro do laço principal, garantindo que o usuário possa interagir com o programa até decidir encerrá-lo.

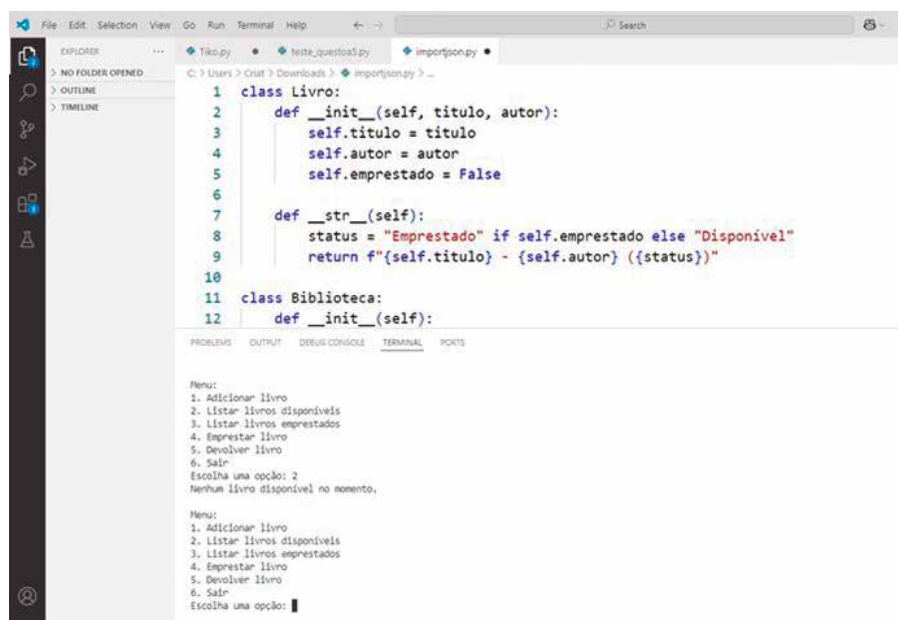
No bloco principal do programa, um objeto da classe `Biblioteca` é criado e armazenado na variável `biblioteca`. O laço `while` garante que o programa continue em execução até que o usuário escolha a opção de sair. Em cada iteração, o menu é exibido e o usuário é solicitado a inserir uma opção numérica. O programa converte a entrada para um número inteiro e, por meio de uma estrutura condicional, verifica qual funcionalidade deve ser executada.

Caso a opção escolhida seja 1, o programa solicita o título e o autor do livro, aplicando `strip()` para remover espaços extras antes e depois do texto digitado. Em seguida, o livro é adicionado à biblioteca utilizando o método correspondente. Se a opção for 2, os livros disponíveis são listados, enquanto a opção 3 exibe os livros emprestados. A opção 4 permite ao usuário informar um título para empréstimo, enquanto a opção 5 possibilita a devolução de um livro. Caso a opção escolhida seja 6, o programa exibe uma mensagem de despedida e encerra a execução com o comando `break`. Se o usuário inserir um número que não corresponde a nenhuma opção válida, o programa exibe uma mensagem informando que a entrada foi inválida.

Para evitar que o programa seja interrompido devido a erros de entrada, um bloco `try-except` envolve a conversão do número da opção. Se o usuário inserir um valor que não possa ser convertido para inteiro, como texto ou caracteres especiais, uma exceção `ValueError` será gerada. Nessa situação, o programa exibe uma mensagem informando o erro e solicita uma nova tentativa.

Essa implementação demonstra a utilização de conceitos fundamentais da programação orientada a objetos, como classes, métodos e atributos. Além disso, emprega boas práticas de tratamento de erros e manipulação de listas. O uso de `strip()` garante que espaços indesejados nas entradas do usuário não interfiram no funcionamento do programa. A aplicação de `lower()` permite que os títulos dos livros sejam comparados de maneira insensível a maiúsculas e minúsculas. O laço `for` percorre listas para exibir os livros disponíveis ou emprestados, e a estrutura condicional dentro dos métodos de empréstimo e devolução assegura que as operações ocorram de maneira adequada. Essas características fazem com que o programa seja intuitivo, funcional e capaz de lidar com diferentes cenários de entrada de dados.

Na figura a seguir, apresentaremos um exemplo da execução do programa completo, no ambiente VS Code. Note novamente que o código-fonte foi digitado na parte superior da figura e executado na parte inferior.



```
1 class Livro:
2     def __init__(self, titulo, autor):
3         self.titulo = titulo
4         self.autor = autor
5         self.emprestado = False
6
7     def __str__(self):
8         status = "Emprestado" if self.emprestado else "Disponível"
9         return f"{self.titulo} - {self.autor} ({status})"
10
11 class Biblioteca:
12     def __init__(self):
```

Menu:
1. Adicionar livro
2. Listar livros disponíveis
3. Listar livros emprestados
4. Empréstimo livro
5. Devolver livro
6. Sair
Escolha uma opção: 2
Nenhum livro disponível no momento.

Menu:
1. Adicionar livro
2. Listar livros disponíveis
3. Listar livros emprestados
4. Empréstimo livro
5. Devolver livro
6. Sair
Escolha uma opção: █

Figura 7 – Exemplo da execução do quinto exercício no ambiente VS Code

1.2 Análise de complexidade: introdução à notação Big-O e à análise de eficiência

A análise de complexidade de algoritmos é área fundamental da ciência da computação, que permite avaliar a eficiência de diferentes soluções para um mesmo problema. Por meio da notação Big-O (também conhecida como notação assintótica ou notação de ordem de grandeza), é possível descrever o comportamento do tempo de execução ou do uso de memória de um algoritmo à medida que o tamanho de entrada cresce.

Essa análise auxilia na tomada de decisões sobre qual estratégia de implementação é mais adequada para problemas que demandam alto desempenho ou lidam com grandes quantidades de dados.

O termo comportamento assintótico refere-se à forma como uma função se comporta quando seu parâmetro tende ao infinito. No contexto de algoritmos, deseja-se saber como o tempo de execução (ou uso de memória) varia em função do tamanho da entrada, denotado com frequência por n .

Em geral, não interessa o valor exato de segundos (no caso de tempo) ou bytes (no caso de memória) que o algoritmo consome; interessa mais a taxa de crescimento desses recursos à medida que n aumenta.

Imagine que você quer organizar uma lista de nomes em ordem alfabética. Suponha que ela tenha apenas 10 nomes ($n = 10$), você consegue fazer isso rapidamente, talvez até manualmente. Mas e se ela tiver 100, 1.000 ou 1.000.000 nomes? Aqui, o comportamento assintótico é quando você olha não apenas para o tempo exato que o computador leva para organizar a lista, mas principalmente para como esse tempo cresce conforme aumenta a quantidade de nomes (o tamanho de entrada, n).

Não interessa se, para 10 nomes, ele leva 0,001 segundo, ou 0,0005 segundo. O que importa é a tendência quando os números ficam grandes. Por exemplo, o tempo pode dobrar, triplicar ou crescer de maneira muito mais rápida à medida que n cresce.

Dito de outro modo, o comportamento assintótico quer medir como isso cresce quando aumentamos o tamanho do problema. Por exemplo, para organizar 10 nomes é rápido, mas se quisermos organizar 1.000.000 de nomes, será bem mais demorado. A forma (ou taxa) como esse tempo aumenta quando passamos de 10 para 100, depois para 1.000 e assim por diante, é o que chamamos de comportamento assintótico.

A notação Big-O é a forma mais amplamente utilizada para descrever limites superiores de crescimento. Em termos intuitivos, dizer que um algoritmo A tem complexidade $O(f(n))$ significa que, para entradas suficientemente grandes, o tempo de execução de A não cresce mais rápido do que uma função constante vezes $f(n)$.

Pense em uma situação do dia a dia: você tem que conferir se cada aluno de uma sala (com n alunos) assinou a lista de presença. Você levará um tempo proporcional ao número de alunos, pois precisa verificar um por um. Se a sala tem 10 alunos, você faz 10 verificações; se tem 100, faz 100 verificações e assim por diante. Esse é um exemplo de comportamento linear: conforme o número de alunos n cresce, o trabalho sobe de forma proporcional e por isso dizemos que a tarefa está em $O(n)$. Usando a definição de Big-O:

- $A(n)$ é o tempo para conferir todas as assinaturas.
- $f(n)$ pode ser simplesmente n .

A afirmação $A(n)=O(n)$ quer dizer que, a partir de um certo ponto (quando n fica grande o bastante), o tempo não crescerá mais depressa do que uma constante vezes n . Em termos práticos, isso significa que se dobrar a quantidade de alunos, aproximadamente dobra o tempo gasto conferindo a lista.

Alguns algoritmos têm tempo de execução constante, o que significa que não sofrem variação perceptível à medida que o tamanho da entrada cresce. São classificados como $O(1)$. Um exemplo típico desse comportamento é o acesso a um elemento específico de um array (ou lista) por índice: teoricamente, considera-se que buscar o valor em uma posição já conhecida tem custo fixo, independentemente de quantos elementos existem ao todo.

Em contrapartida, existem algoritmos cujo tempo de execução é logarítmico, especialmente quando o tamanho do problema é reduzido pela metade (ou por um fator constante) a cada passo. A busca binária em uma lista ordenada é um exemplo clássico: a cada comparação, descarta-se metade dos elementos, o que faz a quantidade de passos crescer de forma proporcional a $\log n$.

Algumas soluções são lineares, o que implica verificar cada elemento de uma lista ou realizar um número de operações diretamente proporcional ao tamanho da entrada. A busca por um elemento em uma lista não ordenada, percorrendo-a de ponta a ponta, ilustra bem esse caso, pois, no pior cenário, é preciso comparar o valor desejado com todos os elementos.

Quando o algoritmo tem tempo de execução do tipo $O(n \log n)$, costuma-se dizer que ele é linear-logarítmico. Esse comportamento surge em algoritmos de ordenação eficientes, como Merge Sort e Quick Sort (na média). Em linhas gerais, ocorre uma divisão recorrente do problema, que gera $\log n$ níveis de recursão ou particionamento, mas em cada nível percorre-se a lista inteira, resultando em um produto $n \log n$. Os algoritmos de ordenação serão abordados em detalhes no tópico 4 deste livro-texto.

Determinados métodos apresentam tempo de execução quadrático, $O(n^2)$. Isso é frequente em algoritmos que utilizam laços aninhados, como Bubble Sort e Insertion Sort, em que para cada elemento da lista é preciso percorrer novamente toda a lista (ou uma fração significativa dela). Assim, no pior caso, o número total de comparações ou operações cresce de forma proporcional a $n \times n$.

Há ainda outros exemplos de complexidades mais elevadas, como $O(n^3)$, que surge em algoritmos com tripla aninhada de laços, por exemplo, em métodos básicos de multiplicação de matrizes. Já $O(2^n)$ é frequente em algoritmos de força bruta, que testam todas as combinações possíveis para resolver um problema de decisão. $O(n!)$ aparece em problemas que demandam analisar todas as permutações de um conjunto (como certas abordagens do problema do caixeiro viajante).

O gráfico da figura 8 compara diferentes complexidades assintóticas típicas da análise de algoritmos.

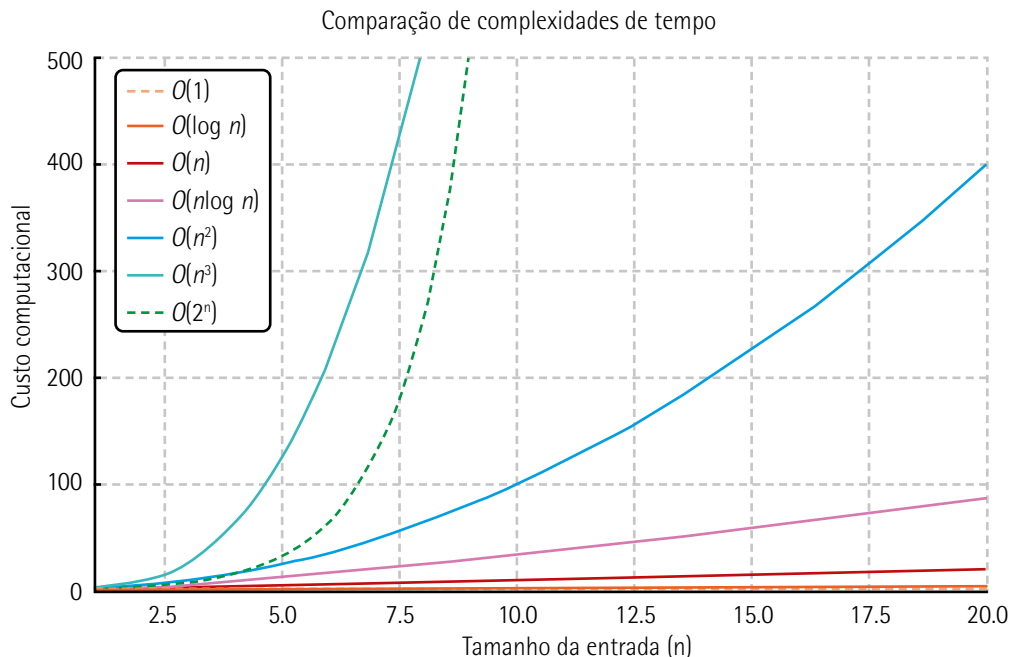


Figura 8 – Análise de algoritmos usando Big-O: comparando as diferentes complexidades assintóticas típicas. $O(n!)$ não está representado porque cresce rápido demais para ser útil nessa visualização

A análise de complexidade é fundamental por diversas razões. Em primeiro lugar, ela permite prever se um algoritmo será escalável para grandes valores de n , algo essencial em aplicações com grandes volumes de dados. Em segundo lugar, possibilita comparar diferentes algoritmos para o mesmo problema, ajudando a escolher a solução de melhor desempenho no contexto desejado.

Além disso, é útil para avaliar eficiência de recursos, pois a análise não se limita apenas ao tempo, mas pode abranger o uso de memória. Por fim, o processo de análise costuma indicar caminhos para otimização, levando ao refinamento de soluções para torná-las mais eficientes.

Também é importante observar que muitos algoritmos apresentam variação de comportamento de acordo com a natureza da entrada. Isso gera a distinção entre melhor caso, caso médio e pior caso. O Quick Sort, por exemplo, tem melhor caso e caso médio em $O(n \log n)$, mas pode se tornar $O(n^2)$ em situações desfavoráveis, quando a partição dos dados é altamente desequilibrada em sucessivas chamadas recursivas.

No uso prático, nem sempre o algoritmo assintoticamente mais rápido é o mais indicado. Para entradas pequenas, métodos $O(n^2)$ podem superar outros $O(n \log n)$ devido a fatores como constantes menores de tempo ou melhor uso de cache. É por isso que engenheiros e cientistas da computação realizam não apenas análise teórica, mas testes empíricos (microbenchmarks) para embasar suas decisões. Em alguns cenários, por exemplo, pode-se preferir Insertion Sort a Merge Sort quando a lista é muito pequena, seja pela implementação mais simples, seja porque o algoritmo pode fazer poucas comparações se os elementos já estiverem quase ordenados.

Daremos um exemplo cotidiano da análise de algoritmos. Imagine que você trabalha em uma grande empresa e precisa verificar, frequentemente, se uma pessoa específica está na lista de funcionários. Existem várias maneiras de resolver esse problema, cada uma levando um tempo diferente conforme a lista cresce. Detalharemos a seguir três possibilidades.

Lista fixa com índice direto (acesso constante, $O(1)$)

- Suponha que exista um sistema digital em que cada funcionário receba um código único (por exemplo, um ID numérico). Você usa esse código como índice em um mapa ou tabela de dados. Assim que digita o código, o sistema vai direto ao registro correspondente.
- Esse método não depende do tamanho da lista de funcionários, a verificação é sempre praticamente imediata (por isso, chamamos de tempo constante).

Busca binária em lista ordenada (tempo logarítmico, $O(\log n)$)

- Agora, pense que você tem a lista de nomes de funcionários ordenada alfabeticamente em um arquivo. Para encontrar alguém, você abre o arquivo na metade e compara para saber se o nome procurado vem antes ou depois na ordem alfabética. Em seguida, descarta a metade que não interessa e repete o processo na metade restante.
- A cada passo, você reduz o universo de busca pela metade. Isso faz com que o tempo de procura cresça de forma logarítmica em relação à quantidade de nomes na lista.

Percorrendo uma lista sem ordem (tempo linear, $O(n)$)

- Por fim, imagine que você tem apenas uma lista simples, sem qualquer ordenação e sem um sistema de acesso por índices. Para descobrir se alguém está na lista, é preciso percorrê-la do primeiro até o último nome, conferindo um por um.
- Nesse caso, se a lista tiver n funcionários, no pior cenário, você examinará os n nomes, tornando o tempo de busca proporcional a n .

No dia a dia, essas abordagens podem ser encontradas em diferentes formas de busca ou verificação de informação. A escolha depende da estrutura de dados disponível, da forma como você tem acesso aos registros e de quantos funcionários (ou itens) precisa manipular.

Uma vez que o Big-O é uma forma de medir e comparar a eficiência de algoritmos ao analisar como o tempo de execução (ou uso de memória) cresce conforme aumenta o tamanho da entrada, ele nos permite:

- **Comparar algoritmos:** se um algoritmo A tem complexidade $O(n)$ e outro algoritmo B tem complexidade $O(n^2)$, isso indica que, à medida que n cresce, B tende a ficar mais lento muito mais rapidamente do que A – mesmo que, para tamanhos pequenos de n , B possa até ser rápido.

- **Estimar escalabilidade:** a notação Big-O revela se um algoritmo será viável ou inviável para grandes volumes de dados. Por exemplo, um algoritmo em $O(2^n)$ pode até funcionar rapidamente para n pequeno, mas se torna praticamente impossível de rodar quando n é grande, devido à explosão combinatória.
- **Identificar pontos de melhora:** muitas vezes, o processo de analisar a complexidade ajuda a enxergar onde o algoritmo está gastando mais tempo. Isso pode guiar otimizações, por exemplo, transformando um método $O(n^2)$ em outro $O(n \log n)$ ao reorganizar laços ou usar estruturas de dados mais adequadas.
- **Ter uma linguagem padrão de descrição:** programadores, cientistas de dados e engenheiros podem se comunicar melhor ao descrever algoritmos. Dizer que um determinado método de busca é $O(n \log n)$ transmite imediatamente a outra pessoa que o tamanho do problema é reduzido a cada passo, sem precisar entrar nos detalhes de implementação.

Portanto, ao usar a notação Big-O, você ganha uma visão mais clara sobre como o algoritmo cresce com o tamanho da entrada, direcionando tanto o desenho de soluções mais eficientes como decisões práticas na hora de lidar com grandes quantidades de dados.

2 ESTRUTURAS DE DADOS LINEARES

As estruturas de dados lineares recebem essa denominação porque organizam os elementos em uma sequência ordenada, na qual cada item possui um predecessor e um sucessor (exceto o primeiro e o último). Essa disposição implica que os elementos são acessados de maneira sequencial, respeitando a ordem na qual foram inseridos. Diferentemente das estruturas não lineares, como árvores e grafos, que permitem múltiplas conexões entre os elementos, as estruturas lineares mantêm uma relação direta entre os itens, o que facilita a manipulação e o processamento dos dados.

A linearidade dessas estruturas permite a realização de operações como pesquisa, inserção e remoção com base na posição dos elementos. Em listas e arrays, por exemplo, o acesso a um item pode ser feito por meio de um índice numérico, garantindo eficiência quando a posição do elemento é conhecida. No entanto, a inserção e a remoção em determinadas posições podem exigir deslocamentos de outros elementos, o que impacta o desempenho dependendo da implementação utilizada.

Além da organização sequencial, pilhas e filas são classificadas como estruturas lineares porque seguem uma ordem específica de inserção e remoção. Embora a maneira como os elementos são acessados seja restrita por regras como LIFO (Last In, First Out) para pilhas e FIFO (First In, First Out) para filas, a estrutura subjacente continua sendo linear, pois os elementos são armazenados em uma única sequência contínua.

A principal vantagem das estruturas lineares é a simplicidade na implementação e na manipulação dos dados. Como os elementos são acessados em uma sequência definida, as operações podem ser otimizadas para determinadas situações, como busca direta em arrays ou remoção eficiente em filas duplamente encadeadas. Entretanto, sua organização também impõe limitações, especialmente quando se trata da necessidade de reorganizar elementos para permitir inserções ou remoções em posições específicas.

A escolha por estruturas lineares deve levar em conta o tipo de acesso aos dados e a complexidade das operações necessárias. Quando há necessidade de relacionamentos hierárquicos entre elementos ou múltiplas conexões, como acontece em árvores e grafos, as estruturas não lineares tornam-se mais apropriadas. Contudo, para armazenar e processar dados de forma sequencial, as estruturas lineares continuam sendo uma das abordagens mais eficientes e amplamente utilizadas na computação.

As estruturas de dados lineares desempenham papel fundamental no desenvolvimento de algoritmos e na organização eficiente de informações em programas. Em Python, listas e arrays são amplamente utilizados para armazenar e manipular conjuntos de elementos, permitindo operações como pesquisa, inserção e remoção. Além dessas estruturas, pilhas e filas oferecem abordagens específicas para gerenciar dados de maneira ordenada, sendo aplicadas em diversas situações na computação. O conhecimento sobre essas estruturas é essencial para otimizar o desempenho de algoritmos e garantir a eficiência do processamento de dados.

As listas em Python são algumas das estruturas mais versáteis da linguagem, pois permitem armazenar elementos de diferentes tipos e realizar operações variadas com facilidade. A pesquisa de elementos pode ser feita utilizando o operador `in` ou métodos como `.index()`, que retorna a posição do item desejado. A inserção pode ser realizada com o método `.append()`, que adiciona elementos ao final da lista, ou com `.insert()`, que insere um item em uma posição específica. Já a remoção pode ocorrer com `.remove()`, caso o elemento seja conhecido, ou `.pop()`, que remove e retorna um item de um índice determinado. Embora listas sejam flexíveis e eficientes para muitos cenários, seu desempenho pode ser impactado quando há operações frequentes de inserção e remoção em posições intermediárias, exigindo a realocação de elementos.

Os arrays, disponíveis no módulo `array` da biblioteca padrão de Python, são uma alternativa às listas quando se deseja uma estrutura mais eficiente para armazenar elementos de um único tipo. Diferentemente das listas, os arrays exigem a definição explícita do tipo de dado que será armazenado, o que reduz o consumo de memória e melhora o desempenho em certas operações. A manipulação de arrays segue uma lógica semelhante à das listas, mas sua aplicação é mais comum em cenários que demandam maior controle sobre a alocação de memória, como em cálculos numéricos intensivos e processamento de grandes volumes de dados.

Além de listas e arrays, pilhas e filas são estruturas que seguem regras específicas de organização e acesso aos elementos. Pilhas operam com a política LIFO, na qual o último elemento inserido é o primeiro a ser removido. Em Python, a implementação de uma pilha pode ser feita utilizando listas, com os métodos `.append()` para inserção e `.pop()` para remoção. As filas, por outro lado, seguem a política FIFO, na qual os elementos são removidos na ordem em que foram adicionados. A implementação de filas pode ser realizada com a estrutura `deque` do módulo `collections`, que oferece operações eficientes de inserção e remoção em ambas as extremidades.

A escolha entre listas, arrays, pilhas e filas depende do contexto da aplicação e das operações que serão executadas com maior frequência. Listas são adequadas para armazenamento genérico de elementos heterogêneos, enquanto arrays são preferíveis para manipulação eficiente de dados homogêneos. Pilhas são úteis em problemas que exigem rastreamento de estados anteriores, como a execução de

chamadas recursivas e o gerenciamento de histórico de navegação. Filas, por sua vez, são empregadas em sistemas que necessitam de processamento sequencial de requisições, como filas de impressão e algoritmos de escalonamento. A compreensão dessas estruturas e suas características possibilita a construção de soluções mais eficazes e bem desenvolvidas.

2.1 Listas e arrays: manipulação, pesquisa, inserção e remoção

A aprendizagem de Python pode ser mais eficaz quando os conceitos teóricos são apresentados de forma aplicada, por meio de exercícios práticos que reproduzem desafios reais do mercado. Em vez de estudar estruturas de dados lineares de maneira abstrata, conduziremos o aprendizado com exemplos concretos extraídos de áreas como Big Data, IoT (Internet das Coisas), computação em nuvem e cibersegurança. Dessa forma, listas e arrays deixam de ser apenas elementos sintáticos da linguagem para se tornarem ferramentas essenciais na manipulação de grandes volumes de dados, processamento de séries temporais, armazenamento distribuído e análise de registros de acesso.

Ao trabalhar com listas e arrays em cenários reais, a experiência adquirida vai além da simples implementação desses conceitos, desenvolvendo também habilidades analíticas e de resolução de problemas. A organização e o tratamento de dados de sensores e dispositivos conectados, por exemplo, exigem a aplicação eficiente dessas estruturas para possibilitar análises preditivas em Big Data. Da mesma forma, o armazenamento e a recuperação de informações em bancos NoSQL dependem do uso adequado dessas técnicas para garantir desempenho e escalabilidade. No campo da cibersegurança, a análise de logs de acesso permite a identificação de atividades suspeitas, demandando abordagens específicas para a extração e o processamento desses dados.

Big Data e IoT são áreas interconectadas que transformam a maneira como capturamos e analisamos dados do mundo real. Big Data envolve o tratamento e a análise de volumes imensos de informações, muitas vezes oriundos de fontes diversas, com o objetivo de extrair insights que apoiem a tomada de decisões estratégicas em organizações. Ele possibilita que empresas identifiquem padrões, prevejam tendências e otimizem processos operacionais, proporcionando uma vantagem competitiva em um mercado cada vez mais orientado por dados.



Observação

O termo insight refere-se a uma compreensão significativa obtida a partir da análise dos dados. Tais compreensões vão além da simples visualização de informações, pois envolvem a identificação de padrões, tendências e correlações que podem não ser imediatamente perceptíveis. No caso do Big Data, esses insights são extraídos por meio de técnicas avançadas, como aprendizado de máquina, estatística e modelagem preditiva, permitindo que organizações tomem decisões mais embasadas, otimizem processos, reduzam riscos e identifiquem novas oportunidades de negócio.

A IoT, por sua vez, refere-se à interconexão de dispositivos físicos equipados com sensores, que coletam e transmitem dados continuamente. Esses dispositivos podem variar de equipamentos industriais e veículos a aparelhos domésticos, possibilitando uma visão em tempo real do ambiente monitorado.

A integração dos dados provenientes desses sensores com tecnologias de Big Data permite uma análise detalhada e imediata, ajudando a identificar anomalias, prevenir falhas e promover manutenções preditivas. Assim, tanto a IoT quanto o Big Data se tornam essenciais para sistemas inteligentes que precisam reagir rapidamente a mudanças e garantir a eficiência operacional.

Neste primeiro exercício, simularemos um ambiente de monitoramento em uma fábrica inteligente, na qual milhares de sensores medem a temperatura de equipamentos e áreas críticas. Essa simulação permitirá que se explore a manipulação de listas em Python para armazenar e processar esses dados. Com essa prática, o leitor aprenderá a gerar dados aleatórios, filtrar leituras críticas, inserir novos dados conforme sensores são adicionados e remover valores anômalos, garantindo a integridade do conjunto de dados para futuras análises e diagnósticos preditivos.

Exemplo de aplicação

Exemplo 1 – Big Data e IoT

Imagine que você atua como desenvolvedor em uma fábrica inteligente, na qual cada máquina e setor crítico são monitorados por sensores de temperatura. Os sensores são responsáveis por coletar dados continuamente, com leituras ocorrendo a cada segundo, e esses dados são enviados para um sistema centralizado.

O sistema precisa realizar diversas operações: inicialmente, gerar um grande volume de leituras simuladas para representar a situação real; depois, filtrar as leituras que indiquem temperaturas acima de 30 °C, pois esses valores podem sinalizar possíveis falhas ou riscos de superaquecimento; em seguida, permitir a inserção de uma nova leitura, simulando a ativação ou atualização de um sensor; e, finalmente, remover dados que estejam fora do intervalo operacional confiável (abaixo de 15 °C ou acima de 35 °C), garantindo que somente informações relevantes e precisas sejam consideradas.

As regras de negócio exigem que o sistema trate rapidamente os dados para alertar a equipe de manutenção e possibilitar intervenções imediatas, além de manter um histórico limpo para análises preditivas.

Utilizaremos listas em Python, uma das estruturas de dados mais versáteis da linguagem, que permite armazenar sequências de elementos de maneira dinâmica. Ao trabalhar com listas, é possível aplicar diversas operações essenciais, como inserção, remoção e filtragem de dados. Esse exercício foca especialmente na utilização de list comprehensions, que fornecem uma forma concisa e eficiente para a criação e a manipulação de listas, permitindo a construção de soluções com menos código e maior clareza.

A modularização do código por meio da definição de funções é outro conceito fundamental que será explorado. Ao segmentar o código em funções específicas – por exemplo, para gerar dados, filtrar

temperaturas, inserir novas leituras e remover dados anômalos –, facilitamos tanto a manutenção quanto a escalabilidade do sistema. Essa abordagem modular é crucial em cenários de Big Data e IoT, nos quais diferentes partes do sistema podem ser atualizadas ou substituídas sem afetar o funcionamento global. Além disso, a utilização de funções promove a reutilização do código, tornando o desenvolvimento mais ágil e seguro.

Outra dimensão relevante é o tratamento de dados simulados, que serve para representar, de forma prática, como os dados reais seriam coletados em um ambiente IoT. Através da geração de números aleatórios dentro de um intervalo definido, simulamos leituras de sensores que variam conforme condições reais.

Esse método, embora simples, é uma aproximação eficaz para testar e validar algoritmos de processamento de dados, permitindo experimentar operações de filtragem, inserção e limpeza de dados antes de lidar com sistemas de produção em larga escala. A seguir consta o código-fonte em Python que implementa o cenário descrito anteriormente.

```
import random

def gerar_dados_sensor(qtd_leituras):
    """
    Gera uma lista de leituras simuladas de temperatura.
    Cada leitura é um valor aleatório entre 15.0°C e 35.0°C, arredondado para
    duas casas decimais.
    """
    dados_sensor = [round(random.uniform(15.0, 35.0), 2) for _ in range(qtd_
    leituras)]
    return dados_sensor

def filtrar_temperaturas_altas(dados_sensor, limite=30.0):
    """
    Filtra e retorna as leituras cuja temperatura ultrapassa o valor limite.
    """
    leituras_altas = [temperatura for temperatura in dados_sensor if temperatura
    > limite]
    return leituras_altas

def inserir_leitura_sensor(dados_sensor, nova_leitura, posicao=None):
    """
    Insere uma nova leitura na lista de dados do sensor.
    Se a posição não for especificada ou for inválida, a nova leitura é
    adicionada ao final da lista.
    """
    if posicao is None or posicao >= len(dados_sensor):
        dados_sensor.append(nova_leitura)
    else:
        dados_sensor.insert(posicao, nova_leitura)
    return dados_sensor

def remover_leituras_anormais(dados_sensor, limite_inferior=15.0, limite_
superior=35.0):
    """
```

Remove da lista de dados as leituras que estejam fora do intervalo definido pelos limites inferior e superior.

```
"""
dados_limpos = [temperatura for temperatura in dados_sensor if limite_
inferior <= temperatura <= limite_superior]
return dados_limpos

def principal():
    quantidade_leituras = 10000 # Número de leituras simuladas
    dados = gerar_dados_sensor(quantidade_leituras)
    print(f"Quantidade de leituras geradas: {len(dados)}")

    leituras_com_temperatura_alta = filtrar_temperaturas_altas(dados,
limite=30.0)
    print(f"Quantidade de leituras acima de 30°C: {len(leituras_com_temperatura_
alta)}")

    nova_leitura = 28.5 # Simula a adição de um novo sensor ou atualização de
leitura
    dados = inserir_leitura_sensor(dados, nova_leitura)
    print(f"Quantidade de leituras após inserção: {len(dados)}")

    dados_limpos = remover_leituras_anormais(dados, limite_inferior=15.0,
limite_superior=35.0)
    print(f"Quantidade de leituras após remoção de anomalias: {len(dados_
limpos)}")

if __name__ == "__main__":
    principal()
```

O quadro 1 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 1 – Funções usadas no código-fonte do primeiro exercício prático

Função	Descrição
random.uniform(a, b)	Gera um número de ponto flutuante aleatório no intervalo entre 'a' e 'b'
round(numero, ndigits)	Arredonda o número para 'ndigits' casas decimais
len(objeto)	Retorna o número de elementos do objeto (como listas, strings ou dicionários)
append(valor)	Adiciona um elemento ao final de uma lista
insert(posicao, valor)	Insere um elemento na posição especificada dentro da lista
print(valor)	Exibe a saída no console
if name == 'main':	Garante que o bloco de código dentro de principal() seja executado apenas quando o script for executado diretamente, e não quando importado como módulo

O código apresentado inicia com a importação do módulo `random`, que é fundamental para a simulação das leituras de sensores. A função `gerar_dados_sensor` é responsável por criar uma lista com um número específico de leituras, utilizando uma list comprehension que gera valores aleatórios entre 15.0°C e 35.0°C e os arredonda para duas casas decimais. Essa função simula a coleta de dados de sensores distribuídos pela fábrica, possibilitando a replicação de um ambiente real de Big Data e IoT.

Em seguida, a função `filtrar_temperaturas_altas` recebe a lista de leituras e emprega uma list comprehension para selecionar apenas os valores que ultrapassam um determinado limite – neste caso, 30°C. Essa filtragem é essencial para identificar potenciais situações críticas que demandam intervenções imediatas. Posteriormente, a função `inserir_leitura_sensor` é utilizada para adicionar uma nova leitura à lista, seja na última posição, caso nenhuma posição específica seja informada, ou em um índice definido, simulando a atualização ou expansão do conjunto de sensores. Finalmente, a função `remover_leituras_anormais` percorre a lista de dados e remove as leituras que estejam fora do intervalo considerado normal (abaixo de 15°C ou acima de 35°C), garantindo a integridade dos dados que serão utilizados para análises preditivas e diagnósticos de falhas.

A função `principal` integra todas as operações, iniciando com a geração de 10.000 leituras simuladas, o que reflete um ambiente de alta frequência de coleta de dados. Em seguida, o código filtra as leituras com temperatura acima de 30 °C, insere uma nova leitura e, por fim, remove os dados que não se enquadram nos parâmetros operacionais definidos.

A seguir, apresentaremos algumas sugestões para implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Para aprimorar mais essa solução, pode-se integrar bibliotecas especializadas como o NumPy, que otimiza o processamento de grandes volumes de dados por meio de arrays multidimensionais, proporcionando melhor desempenho computacional.
- A incorporação do Pandas também é vantajosa, permitindo a transformação dos dados em DataFrames e facilitando a realização de análises estatísticas e a visualização de tendências através de gráficos.
- Outra melhoria relevante é a implementação de um sistema de processamento em tempo real, utilizando protocolos como MQTT ou WebSocket, para que os dados dos sensores sejam processados conforme são gerados, possibilitando respostas imediatas a anomalias.
- A persistência dos dados em bancos de dados NoSQL, como o MongoDB, e a adição de mecanismos robustos de tratamento de exceções e validações, assegura uma solução mais resiliente e escalável para ambientes industriais reais.

Agora vamos para o segundo exercício. No contexto atual, o processamento de séries temporais e logs tem se tornado uma ferramenta indispensável para análises preditivas, sobretudo em ambientes em que grandes volumes de dados são gerados continuamente, como em aplicações de Big Data e IoT. Esses dados, muitas vezes coletados por sensores e dispositivos conectados, permitem identificar padrões e tendências que podem antecipar problemas ou indicar oportunidades de otimização. Ao compreender e processar esses registros, é possível extrair informações valiosas para tomadas de decisão e intervenções preventivas, proporcionando um ganho significativo em eficiência e segurança operacional.

A importância desse tema se reflete no fato de que as séries temporais representam a evolução de variáveis ao longo do tempo, o que é crucial para prever comportamentos futuros e detectar anomalias. Logs gerados por sistemas, dispositivos e sensores acumulam um histórico que, quando analisado, pode revelar picos de desempenho, quedas abruptas ou variações inesperadas. Essa análise preditiva possibilita a criação de sistemas inteligentes que monitoram e antecipam falhas ou eventos críticos, permitindo uma resposta proativa, em vez de reativa, diante de problemas iminentes.

O processamento desses dados é a base para a implementação de soluções robustas em diversas áreas, como manutenção preditiva, monitoramento ambiental e otimização de processos industriais.

Exemplo de aplicação

Exemplo 2 – Big Data e IoT

Imagine que você trabalha para uma empresa que administra um parque de servidores e dispositivos IoT distribuídos em diversas instalações industriais. Cada dispositivo gera um log contendo uma marca temporal e um valor de desempenho ou temperatura, registrados a cada minuto.

O sistema central precisa processar esses logs para monitorar a saúde dos equipamentos e antecipar possíveis falhas.

Entre os requisitos do sistema, destaca-se a necessidade de gerar um grande volume de logs simulados, calcular a média móvel em intervalos de cinco minutos para identificar tendências, detectar anomalias quando a média ultrapassa um determinado limite (indicando, por exemplo, um superaquecimento iminente) e permitir a inserção de novos logs à medida que os dispositivos são atualizados ou novos dados são recebidos.

As regras de negócio exigem que o sistema trate os dados com rapidez e precisão, mantendo um histórico confiável para análises preditivas que possam evitar interrupções críticas e reduzir custos operacionais.

Neste exercício, exploramos os recursos nativos de Python para criação, manipulação e análise de listas, aplicando-os ao processamento de séries temporais e logs. Novamente, um dos principais recursos utilizados é a list comprehension, que permite gerar listas de forma concisa e eficiente.

Por exemplo, na função que gera os logs, usamos uma list comprehension para iterar um número determinado de vezes e criar, para cada iteração, um dicionário contendo um timestamp e um valor numérico aleatório. Esse recurso não somente torna o código mais compacto, como melhora a legibilidade e a performance, eliminando a necessidade de estruturas de loop mais verbosas.

Outro recurso fundamental empregado é o uso do slicing (fatiamento) de listas, que possibilita acessar sublistas de forma rápida e intuitiva. Essa técnica é especialmente útil para calcular a média móvel, em que selecionamos uma janela de registros consecutivos usando a sintaxe `logs[i:i+janela]`. Conjuntamente à função nativa `sum`, que agrega os valores dessa sublista, e à função `round`, que

arredonda os resultados para duas casas decimais, o slicing facilita o processamento de intervalos específicos dos dados.

Adicionalmente, o módulo `datetime` é utilizado para manipular datas e horas, permitindo a criação e o gerenciamento de timestamps para cada log gerado. Com funções como `datetime.now()` e a classe `timedelta`, é possível somar intervalos e gerar sequências temporais que simulam leituras periódicas de sensores. Essa capacidade de manipulação temporal é crucial para o processamento de séries temporais, pois permite ordenar e comparar registros com precisão.

Ao integrar esses recursos – list comprehension, slicing, funções nativas de agregação e módulo `datetime` –, o exercício demonstra como construir soluções robustas para a análise preditiva baseada em logs, combinando eficiência, clareza e praticidade na implementação com Python. A seguir, consta o código-fonte da solução:

```
import random
from datetime import datetime, timedelta

def gerar_logs_temporais(qtd_logs, inicio, intervalo_segundos):
    """
    Gera uma lista de logs simulados, onde cada log possui um timestamp e um
    valor numérico.
    Cada log representa uma leitura realizada a cada 'intervalo_segundos',
    iniciando em 'inicio'.
    O valor é um número aleatório entre 20.0 e 80.0, arredondado para duas casas
    decimais.
    """
    logs = []
    tempo_atual = inicio
    for _ in range(qtd_logs):
        valor = round(random.uniform(20.0, 80.0), 2)
        log = {"tempo": tempo_atual, "valor": valor}
        logs.append(log)
        tempo_atual += timedelta(seconds=intervalo_segundos)
    return logs

def calcular_media_movel(logs, janela):
    """
    Calcula a média móvel para uma janela de tamanho 'janela' sobre a lista de
    logs.
    Retorna uma lista de dicionários com o timestamp correspondente à última
    leitura da janela e a média calculada para aquele intervalo.
    """
    medias_moveis = []
    for i in range(len(logs) - janela + 1):
        soma = sum(log["valor"] for log in logs[i:i+janela])
        media = soma / janela
        registro_media = {"tempo": logs[i + janela - 1]["tempo"], "media":
round(media, 2)}
        medias_moveis.append(registro_media)
    return medias_moveis
```

```
def inserir_log(logs, novo_log, posicao=None):
    """
    Insere um novo log na lista 'logs'.
    Se a posição não for especificada ou for maior que o tamanho atual da lista,
    o log é adicionado ao final.
    """
    if posicao is None or posicao >= len(logs):
        logs.append(novo_log)
    else:
        logs.insert(posicao, novo_log)
    return logs

def detectar_anomalias(medias_moveis, limite):
    """
    Identifica anomalias na lista de médias móveis, retornando os registros cuja
    média ultrapassa o limite especificado.
    """
    anomalias = [registro for registro in medias_moveis if registro["media"] >
limite]
    return anomalias

def principal():
    inicio = datetime.now()
    qtd_logs = 100 # Número de logs simulados
    intervalo_segundos = 60 # Intervalo de 60 segundos entre cada log
    logs = gerar_logs_temporais(qtd_logs, inicio, intervalo_segundos)

    print("Exibindo os primeiros 5 logs gerados:")
    for log in logs[:5]:
        print(f"{log['tempo']} - Valor: {log['valor']}")

    janela = 5 # Tamanho da janela para o cálculo da média móvel
    medias_moveis = calcular_media_movel(logs, janela)

    print("\nExibindo as primeiras 5 médias móveis calculadas:")
    for media in medias_moveis[:5]:
        print(f"{media['tempo']} - Média: {media['media']}")

    limite_anomalia = 70.0 # Limite acima do qual uma média móvel é considerada
uma anomalia
    anomalias = detectar_anomalias(medias_moveis, limite_anomalia)
    print(f"\nQuantidade de anomalias detectadas (média móvel > {limite_
anomalia}): {len(anomalias)}")

    # Inserindo um novo log simulando a atualização dos dados
    nova_data = logs[-1]["tempo"] + timedelta(seconds=intervalo_segundos)
    novo_valor = round(random.uniform(20.0, 80.0), 2)
    novo_log = {"tempo": nova_data, "valor": novo_valor}
    logs = inserir_log(logs, novo_log)
    print(f"\nNovo log inserido: {novo_log['tempo']} - Valor: {novo_
log['valor']}")
    print(f"Total de logs após inserção: {len(logs)}")

if __name__ == "__main__":
    principal()
```


O quadro 2 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas no quadro 1 foram omitidas para evitar duplicidade.

Quadro 2 – Funções usadas no código-fonte do segundo exercício prático

Função	Descrição
<code>datetime.now()</code>	Retorna a data e a hora atuais do sistema
<code>timedelta(seconds=n)</code>	Cria um intervalo de 'n' segundos, utilizado para manipulação de datas e horários
<code>sum(iterável)</code>	Calcula a soma dos elementos de um iterável (como listas e dicionários)
<code>for elemento in iterável</code>	Itera sobre os elementos de um iterável, executando um bloco de código para cada item
<code>dicionario[chave]</code>	Acessa o valor correspondente a uma chave dentro de um dicionário
<code>dicionario['chave'] = valor</code>	Atribui um valor a uma chave específica dentro de um dicionário

O código inicia com a importação dos módulos necessários, na qual o módulo `random` é utilizado para gerar valores numéricos aleatórios. Os módulos `datetime` e `timedelta` são empregados para manipular datas e horários, permitindo a criação de timestamps para os logs simulados.

A função responsável por gerar os logs, denominada `gerar_logs_temporais`, recebe como parâmetros a quantidade de logs a ser gerada, o instante de início e o intervalo em segundos entre cada leitura. Essa função cria uma lista de dicionários, na qual cada um deles representa um log com uma chave `tempo` que armazena o timestamp e uma chave `valor` que armazena a leitura numérica, simulada como um valor aleatório entre 20.0 e 80.0.



Observação

O uso do underscore (`_`) como variável no `for` na linha `for _ in range(qtd_logs):` indica intencionalmente que o valor da variável não tem importância. A iteração está sendo feita apenas pela contagem, não pelo valor em si. Em outras palavras, o loop diz: "repita esse bloco `qtd_logs` vezes, mas não preciso de um nome para o índice ou valor do contador".

Se fosse usada uma variável como `i`, contador ou qualquer outro nome, e essa variável não fosse utilizada, isso indicaria que foi criada desnecessariamente, o que não é ideal em termos de clareza e estilo de código. O underscore evita esse ruído e torna explícito que a variável está ali apenas por exigência da sintaxe do `for`, não por necessidade do valor.

A função `calcular_media_movel` é projetada para processar a lista de logs e calcular a média móvel em uma janela definida pelo usuário. Para cada conjunto de registros consecutivos que formam a janela, o código calcula a soma dos valores e divide pelo número de elementos da janela, armazenando o resultado em um novo dicionário que guarda o timestamp da última leitura da janela e o valor da média, ambos devidamente arredondados. Dessa maneira, obtém-se uma nova lista que reflete a

evolução dos dados de forma suavizada, permitindo identificar tendências e comportamentos que possam indicar anomalias.

O código também inclui funções para inserir novos logs e detectar anomalias. A função `inserir_log` permite adicionar um novo registro à lista, seja em uma posição específica ou ao final, simulando a atualização contínua dos dados conforme novas informações são recebidas.

A função `detectar_anomalias` analisa as médias móveis calculadas e identifica aqueles registros cuja média ultrapassa um limite predefinido, considerado como indicador de comportamento anormal.

Por fim, a função principal orquestra a execução de todas as funções, gerando os logs, calculando as médias móveis, detectando anomalias e inserindo um novo log, além de imprimir os resultados parciais para visualização e validação do processo.

A seguir, apresentaremos algumas sugestões para implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Para expandir e sofisticar esse código, uma abordagem interessante é integrar bibliotecas especializadas como o NumPy e o Pandas, que oferecem funcionalidades avançadas para a manipulação de arrays e DataFrames, possibilitando cálculos mais eficientes e análises estatísticas detalhadas.
- Outra melhoria é implementar um mecanismo de leitura de logs em tempo real, utilizando protocolos como MQTT (Message Queuing Telemetry Transport) ou WebSocket, o que permite a atualização dinâmica dos dados e a resposta imediata a eventos críticos.
- A incorporação de técnicas de visualização, utilizando bibliotecas como Matplotlib ou Plotly, facilita a interpretação dos resultados, permitindo a criação de gráficos interativos que evidenciem tendências e anomalias.
- Por fim, a adição de tratamento de exceções e a validação dos dados de entrada garantem maior robustez e escalabilidade do sistema, tornando-o apto a ser aplicado em ambientes industriais e corporativos de alta complexidade.

Para o terceiro exercício, falaremos da computação em nuvem que tem transformado a forma como as empresas armazenam e manipulam dados, permitindo o processamento escalável e a distribuição de informações em larga escala.

Esse paradigma possibilita o acesso a recursos computacionais de forma on-demand, eliminando a necessidade de infraestrutura local robusta. Em particular, bancos NoSQL e sistemas distribuídos emergem como soluções essenciais para lidar com a variabilidade e o volume crescente de dados, oferecendo flexibilidade na modelagem e na consulta de informações que não se encaixam perfeitamente em esquemas relacionais tradicionais.



Observação

NoSQL é um termo usado para designar um conjunto de sistemas de gerenciamento de banco de dados que não segue o modelo tradicional dos bancos relacionais, como o MySQL ou o PostgreSQL. A sigla, que originalmente significava "*not only SQL*" ("não apenas SQL"), indica que esses sistemas não utilizam exclusivamente a linguagem SQL para manipulação de dados, embora alguns ofereçam suporte parcial a ela. O principal diferencial dos bancos NoSQL está no modo como os dados são armazenados e recuperados, oferecendo maior flexibilidade para determinadas aplicações.

Ao contrário dos bancos relacionais, que organizam os dados em tabelas com linhas e colunas fixas, os bancos NoSQL adotam estruturas mais dinâmicas. Entre elas, é possível encontrar bancos baseados em documentos, colunas, grafos ou pares chave-valor. Um banco de dados orientado a documentos, por exemplo, armazena os dados em formatos como JSON ou BSON, permitindo que diferentes registros possuam estruturas distintas e se adaptem mais facilmente a mudanças nos requisitos de uma aplicação.

Nesse contexto, os bancos NoSQL são projetados para suportar ambientes com alta demanda por escalabilidade, disponibilidade e desempenho. Eles armazenam dados de forma não estruturada ou semiestruturada, utilizando modelos de documentos, colunas, grafos ou chaves-valor. Essa diversidade permite que empresas de diferentes setores, como redes sociais, e-commerce e IoT, processem e analisem grandes volumes de dados em tempo real.

Sistemas distribuídos, por sua vez, garantem que essas informações sejam replicadas e gerenciadas em vários servidores, aumentando a resiliência e a eficiência dos serviços em nuvem.

A importância do armazenamento e da manipulação de dados em bancos NoSQL e sistemas distribuídos reside na capacidade de oferecer soluções ágeis para problemas de alta complexidade, sem comprometer a performance mesmo em cenários de grande escala. Ao utilizar essas tecnologias, as organizações podem garantir que suas aplicações permaneçam responsivas e seguras, mesmo diante de picos de acesso e grandes volumes de transações.

Exemplo de aplicação

Exemplo 3 – Computação em nuvem

Imagine que você trabalha para uma empresa que opera um serviço de análise de dados para uma rede de varejo global. Essa empresa utiliza um banco NoSQL distribuído para armazenar registros de transações e interações dos clientes, garantindo que os dados estejam disponíveis em tempo real para análise e geração de relatórios.

Cada registro armazenado inclui um identificador único, um timestamp que indica o momento da transação, o valor da compra e o identificador do dispositivo que realizou a operação.

As regras de negócio exigem que o sistema permita: a inserção de novos registros de transações à medida que ocorrem, a filtragem de registros com base em critérios específicos (por exemplo, transações acima de um certo valor ou realizadas por um determinado dispositivo) e a remoção periódica de registros antigos para manter o desempenho e a relevância dos dados.

Essa abordagem garante que os dados analisados sejam sempre recentes e que o sistema se mantenha ágil e escalável, refletindo o comportamento típico de sistemas distribuídos em nuvem.

Neste exercício, exploraremos a manipulação de dados utilizando listas em Python para simular o funcionamento de um banco NoSQL e sistemas distribuídos. Python é uma linguagem poderosa que oferece recursos nativos, como a criação e a manipulação de listas e dicionários, que são ideais para representar coleções de registros.

Neste contexto, utilizaremos listas para armazenar registros e dicionários para modelar cada registro de forma flexível, permitindo que os dados sejam organizados sem a rigidez de um esquema fixo. Essa abordagem reflete a natureza dos bancos NoSQL, na qual a estrutura dos dados pode variar conforme necessário.

Empregaremos list comprehensions para filtrar e transformar os dados de maneira concisa, facilitando operações como a busca por registros específicos ou a remoção de registros antigos. Funções nativas como `append`, `insert` e `pop` serão utilizadas para manipular a lista de registros, simulando operações de inserção e remoção que ocorrem em sistemas distribuídos. Esses recursos do Python simplificam o código, melhoram sua legibilidade e manutenção, aspectos essenciais quando se trabalha com grandes volumes de dados.

Outro aspecto importante abordado é o tratamento de timestamps utilizando o módulo `datetime`, que permite a criação e a manipulação de datas e horários de forma eficiente. Através desse recurso, poderemos simular a geração de registros com marcas temporais, essenciais para qualquer aplicação de análise de dados em nuvem. Combinando esses conceitos, o exercício demonstrará como construir uma solução que imita o comportamento de um banco NoSQL distribuído, integrando operações de inserção, consulta e remoção de dados de maneira coesa e escalável. A seguir, consta o código-fonte da solução:

```
import random
from datetime import datetime, timedelta

def gerar_registros(qtd_registros, inicio, intervalo_segundos):
    """
    Gera uma lista de registros simulados, na qual cada registro é um dicionário
    representando uma transação em um banco NoSQL distribuído. Cada registro contém
    um 'id', 'timestamp', 'valor' e 'id_dispositivo'.
    """
    registros = []
    tempo_atual = inicio
    for i in range(1, qtd_registros + 1):
```

```
registro = {
    "id": i,
    "timestamp": tempo_atual,
    "valor": round(random.uniform(10.0, 500.0), 2),
    "id_dispositivo": random.choice(["disp_001", "disp_002", "disp_003",
"disp_004"])
}
registros.append(registro)
tempo_atual += timedelta(seconds=intervalo_segundos)
return registros

def filtrar_registros_por_valor(registros, valor_minimo):
    """
    Filtra e retorna os registros cuja transação possui um valor maior ou igual
    a 'valor_minimo'.
    """
    registros_filtrados = [reg for reg in registros if reg["valor"] >= valor_
minimo]
    return registros_filtrados

def inserir_registro(registros, novo_registro, posicao=None):
    """
    Insere um novo registro na lista de registros. Se a posição não for
    especificada ou for inválida, o registro é adicionado ao final da lista.
    """
    if posicao is None or posicao >= len(registros):
        registros.append(novo_registro)
    else:
        registros.insert(posicao, novo_registro)
    return registros

def remover_registros_antigos(registros, tempo_limite):
    """
    Remove da lista de registros aqueles cujo timestamp é anterior ao 'tempo_
limite'.
    """
    registros_atualizados = [reg for reg in registros if reg["timestamp"] >=
tempo_limite]
    return registros_atualizados

def principal():
    inicio = datetime.now()
    qtd_registros = 50 # Número de registros simulados
    intervalo_segundos = 30 # Intervalo de 30 segundos entre cada registro

    # Gerar registros simulados
    registros = gerar_registros(qtd_registros, inicio, intervalo_segundos)
    print("Exibindo os primeiros 5 registros gerados:")
    for reg in registros[:5]:
        print(f"ID: {reg['id']} | Timestamp: {reg['timestamp']} | Valor:
{reg['valor']} | Dispositivo: {reg['id_dispositivo']}")

    # Filtrar registros com valor maior ou igual a 300
    valor_minimo = 300.0
    registros_filtrados = filtrar_registros_por_valor(registros, valor_minimo)
    print(f"\nQuantidade de registros com valor >= {valor_minimo}:
```

```
{len(registros_filtrados)}")

# Inserir um novo registro
novo_registro = {
    "id": len(registros) + 1,
    "timestamp": registros[-1]["timestamp"] + timedelta(seconds=intervalo_
segundos),
    "valor": round(random.uniform(10.0, 500.0), 2),
    "id_dispositivo": "disp_005"
}
registros = inserir_registro(registros, novo_registro)
print(f"\nNovo registro inserido: ID: {novo_registro['id']} | Timestamp:
{novo_registro['timestamp']} | Valor: {novo_registro['valor']} | Dispositivo:
{novo_registro['id_dispositivo']}")
print(f"Total de registros após inserção: {len(registros)}")

# Remover registros com timestamp anterior a 2 minutos após o início
tempo_limite = inicio + timedelta(minutes=2)
registros_atualizados = remover_registros_antigos(registros, tempo_limite)
print(f"\nQuantidade de registros após remoção dos antigos (antes de {tempo_
limite}): {len(registros_atualizados)}")

if __name__ == "__main__":
    principal()
```

O quadro 3 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 3 – Funções usadas no código-fonte do terceiro exercício prático

Função	Explicação de uso
range(início, fim)	Gera uma sequência de números inteiros, incluindo o valor inicial e excluindo o valor final. No código, é usado implicitamente no <code>for i in range(1, qtd_registros + 1)</code> para gerar os identificadores dos registros
random.choice(sequência)	Retorna um elemento aleatório de uma sequência, como uma lista. No código, é usado para escolher aleatoriamente um <code>id_dispositivo</code>
registros[-1]	Acessa o último elemento da lista registros
timedelta(minutes=n)	Cria um intervalo 'n' minutos, utilizado para cálculos com datas e horas. Já havia sido usada com segundos, mas aqui há o uso específico de minutos

O código apresentado inicia com a importação dos módulos necessários: o módulo `random` para gerar valores aleatórios e o módulo `datetime` com `timedelta` para a manipulação de datas e horários, fundamentais para simular registros temporais em um ambiente de computação em nuvem. A função `gerar_registros` é responsável por criar uma lista de registros, cada um representado por um dicionário que contém um identificador único, um timestamp que indica o momento da criação, um valor numérico que simula o valor de uma transação e um identificador de dispositivo, escolhido aleatoriamente em uma lista predefinida. Esse método de geração ilustra como dados não estruturados podem ser organizados de forma flexível, refletindo a natureza dos bancos NoSQL.

A função `filtrar_registros_por_valor` aplica uma list comprehension para selecionar registros cuja transação possui um valor igual ou superior a um limiar especificado. Essa operação é crucial

para extrair dados relevantes, permitindo que apenas as transações de maior impacto sejam analisadas. Em seguida, a função `inserir_registro` oferece a capacidade de adicionar um novo registro à lista, simulando a atualização contínua de dados em um sistema distribuído, no qual novos registros chegam constantemente. A função `remover_registros_antigos` utiliza novamente uma `list comprehension` para eliminar registros cujo timestamp seja anterior a um determinado limite, garantindo que a base de dados se mantenha atualizada e com registros pertinentes para análise, assim como ocorre em sistemas que adotam estratégias de expiração ou arquivamento de dados.

A função principal integra todas as operações propostas, começando pela geração de 50 registros simulados com intervalos regulares de 30 segundos. Em seguida, o código exibe os primeiros registros gerados para conferência, filtra os registros com transações de valor elevado, insere um novo registro simulando a chegada de dados de um dispositivo adicional e, por fim, remove os registros considerados antigos com base em um critério temporal. Essa sequência de operações demonstra como as funcionalidades básicas de manipulação de listas e dicionários em Python podem ser utilizadas para simular um ambiente de armazenamento e processamento de dados em bancos NoSQL e sistemas distribuídos, refletindo cenários comuns em computação em nuvem.

A seguir, apresentaremos algumas sugestões para implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- É interessante integrar bibliotecas especializadas como o NumPy e o Pandas, que oferecem capacidades avançadas para o processamento de grandes volumes de dados e a realização de análises estatísticas de forma mais eficiente.
- A implementação de uma interface para acesso em tempo real aos dados, utilizando protocolos como MQTT ou WebSocket, transforma o exercício em uma simulação de sistema distribuído dinâmico, no qual os registros são atualizados continuamente.
- Outra melhoria importante é a inclusão de mecanismos de tratamento de exceções e validação dos dados de entrada, garantindo maior robustez e resiliência ao sistema.
- Por fim, a utilização de técnicas de visualização de dados com bibliotecas como Matplotlib ou Plotly permite a criação de dashboards interativos, facilitando a interpretação dos registros e a identificação de tendências ou anomalias em tempo real.

Para o quarto exercício, falaremos sobre a cibersegurança, uma área que tem ganhado cada vez mais importância no cenário tecnológico atual, pois protege sistemas, redes e dados contra acessos não autorizados, fraudes e outros tipos de ataques maliciosos.

Com o aumento exponencial da quantidade de informações trafegadas e armazenadas digitalmente, torna-se crucial implementar mecanismos de monitoramento e análise que identifiquem comportamentos suspeitos e previnam incidentes antes que causem danos significativos. Algumas das práticas essenciais nesse contexto são o registro e a análise de logs de acesso, que fornecem um histórico detalhado das interações com sistemas e aplicações, servindo como base para a detecção de atividades atípicas e intrusões.

Em um ambiente no qual diversos usuários e dispositivos acessam recursos digitais, os logs de acesso se tornam a pegada digital deixada por cada interação. Essas informações incluem, por exemplo, endereços IP, horários de acesso, nomes de usuários e status das tentativas de login (sucesso ou falha). Através da análise desses registros, é possível identificar padrões de comportamento, como múltiplas tentativas de acesso falhadas provenientes do mesmo endereço IP, que podem indicar tentativas de ataque de força bruta ou outras atividades maliciosas. Assim, a capacidade de monitorar e analisar esses logs é fundamental para que as equipes de segurança possam agir de forma proativa, reforçando as defesas e mitigando riscos.

Além disso, a análise de logs de acesso permite não apenas detectar ameaças, mas gerar relatórios que auxiliam na compreensão do tráfego e no aperfeiçoamento dos controles de segurança. Essa prática é indispensável para conformidade com normas e regulamentos de proteção de dados e para manter a confiança de usuários e clientes. Ao desenvolver soluções para registrar e analisar esses dados com Python, os profissionais podem criar ferramentas personalizadas que atendam às necessidades específicas de suas organizações, integrando automação, inteligência e monitoramento contínuo em um sistema robusto de cibersegurança.

Exemplo de aplicação

Exemplo 4 – Cibersegurança

Imagine que você trabalha na área de segurança da informação de uma grande empresa que gerencia uma plataforma de acesso corporativo. Todos os acessos aos sistemas são registrados em logs que armazenam informações como o endereço IP de origem, o nome de usuário, a data e a hora do acesso e o resultado da tentativa (se foi bem-sucedida ou se houve falha).

O objetivo do sistema é monitorar esses logs para detectar atividades suspeitas, como múltiplas tentativas de login malsucedidas de um mesmo endereço IP, que podem ser indicativas de um ataque de força bruta.

As regras de negócio definem que, caso um determinado IP registre mais de três falhas de acesso em um curto período, esse IP deve ser marcado como suspeito para futuras análises e, eventualmente, bloqueado ou submetido a uma investigação mais detalhada.

Essas regras influenciam diretamente a implementação do código, que precisa ser capaz de gerar logs simulados, filtrar registros com falhas e identificar quais IPs excederam o limite aceitável de tentativas malsucedidas.

Neste exercício exploraremos a adoção de arrays que viabilizam operações vetorizadas, reduzem sobrecarga computacional e favorecem a memória cache, benefícios valiosos quando se lida com grandes quantidades de registros. Cada log será representado em um array estruturado (`numpy.ndarray` com `dtype` composto), contemplando campos para timestamp, endereço IP, usuário e status. Essa abordagem garante tipagem homogênea em cada coluna e mantém a conveniência de acessar campos por nome, enquanto permite aplicar funções nativas de NumPy para seleção, filtragem e agregação.

NumPy é uma biblioteca fundamental para computação científica em Python, projetada para oferecer suporte eficiente a arrays multidimensionais e uma ampla gama de operações matemáticas de alto desempenho. Seu núcleo é a estrutura `ndarray`, que permite a manipulação de grandes volumes de dados numéricos de maneira vetorizada, isto é, sem a necessidade de laços explícitos em Python. Essa característica reduz substancialmente a sobrecarga computacional associada à iteração tradicional, pois as operações são delegadas a rotinas otimizadas em linguagem de nível mais baixo do que Python, como a linguagem C. Além disso, a disposição contígua dos dados na memória favorece o uso eficiente da memória cache do processador, o que resulta em ganho expressivo de desempenho, especialmente quando se trabalha com conjuntos extensos de registros.

No exemplo proposto, a estrutura de dados adotada é um array do tipo `ndarray` com um `dtype` composto, o qual permite definir diferentes campos – como timestamp, endereço IP, usuário e status – em cada elemento do array. Essa organização promove uma tipagem homogênea dentro de cada coluna, evitando a fragmentação e garantindo consistência ao longo do conjunto de dados. Além disso, a possibilidade de acessar campos por nome confere clareza e simplicidade ao código, ao mesmo tempo que mantém a compatibilidade com as funções nativas da biblioteca. Operações como seleção condicional, filtragem por critérios múltiplos e agregações estatísticas podem ser realizadas de maneira direta e eficiente, aproveitando a infraestrutura de processamento vetorial que torna o NumPy uma ferramenta especialmente adequada para análise de dados em larga escala.

Operações centrais, tais como contagem de falhas por endereço IP e inserção de novos registros, tornam-se mais diretas graças à vetorização. A identificação de IPs suspeitos usa `numpy.unique` com `return_counts=True`, eliminando laços explícitos. Inserções adicionais valem-se de `numpy.append`, que devolve um novo array contendo o registro recém-chegado, mantendo imutabilidade em relação ao conjunto anterior. O módulo `datetime` permanece essencial para gerar timestamps precisos, enquanto `numpy.random` oferece aleatoriedade na escolha de IPs, usuários e status. A seguir, consta o código-fonte da solução:

```
import numpy as np
from datetime import datetime, timedelta

def gerar_logs_acesso_array(qtd_logs, inicio, intervalo_segundos):
    dtipo = np.dtype([
        ('timestamp', 'datetime64[ns]'),
        ('ip', 'U15'),
        ('usuario', 'U20'),
        ('status', 'U7')
    ])
    registros = np.empty(qtd_logs, dtype=dtipo)

    lista_ips = np.array(["192.168.1.1", "192.168.1.2",
                          "192.168.1.3", "10.0.0.1", "10.0.0.2"])
    lista_usuarios = np.array(["usuario_a", "usuario_b",
                               "usuario_c", "usuario_d"])
    lista_status = np.array(["sucesso", "falha"])
    pesos_status = np.array([0.8, 0.2])

    tempo_atual = inicio
```

```

for idx in range(qtd_logs):
    registros[idx] = (
        np.datetime64(tempo_atual, 'ns'),
        np.random.choice(lista_ips),
        np.random.choice(lista_usuarios),
        np.random.choice(lista_status, p=pesos_status)
    )
    tempo_atual += timedelta(seconds=intervalo_segundos)
return registros

def detectar_ips_suspeitos_array(registros, limiar_falhas):
    falhas_mask = registros['status'] == 'falha'
    ips_com_falha = registros['ip'][falhas_mask]
    ips_unicos, contagens = np.unique(ips_com_falha, return_counts=True)
    return ips_unicos[contagens >= limiar_falhas]

def inserir_log_array(registros, novo_log):
    dtipo = registros.dtype
    registro_np = np.array(
        (np.datetime64(novo_log['timestamp'], 'ns'),
         novo_log['ip'],
         novo_log['usuario'],
         novo_log['status']),
        dtype=dtipo
    )
    return np.append(registros, registro_np)

def principal():
    inicio = datetime.now()
    qtd_logs = 100
    intervalo_segundos = 30
    registros = gerar_logs_acesso_array(qtd_logs, inicio, intervalo_segundos)

    print("Primeiros cinco logs:\n")
    for log in registros[:5]:
        ts =
log['timestamp'].astype('datetime64[ms]').astype(datetime)
        print(f"Timestamp: {ts} | IP: {log['ip']} | Usuário: {log['usuario']} |
Status: {log['status']}")

    limiar_falhas = 3
    ips_suspeitos = detectar_ips_suspeitos_array(registros, limiar_falhas)
    print(f"\nIPs suspeitos (com {limiar_falhas} ou mais falhas): {ips_suspeitos}")

    # criação do novo registro utilizando numpy.timedelta64
    novo_ts = registros[-1]['timestamp'] + np.timedelta64(intervalo_segundos,
's')
    novo_log = {
        "timestamp": novo_ts,
        "ip": "192.168.1.99",
        "usuario": "usuario_novo",
        "status": "falha"
    }
    registros = inserir_log_array(registros, novo_log)

```

```
print("\nNovo log inserido:")
ts_novo =
novo_log['timestamp'].astype('datetime64[ms]').astype(datetime)
print(f"Timestamp: {ts_novo} | IP: {novo_log['ip']} |
Usuário: {novo_log['usuario']} | Status: {novo_log['status']}")
print(f"Total de logs após inserção: {len(registros)}")

if __name__ == "__main__":
    principal()
```

O quadro 4 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 4 – Funções usadas no código-fonte do quarto exercício prático

Função ou construção	Explicação de uso
<code>np.dtype([...])</code>	Define a estrutura de um tipo composto para arrays NumPy, permitindo armazenar diferentes campos (como um registro com timestamp, IP, usuário e status) em um único array estruturado
<code>np.empty(tamanho, dtype=...)</code>	Cria um array não inicializado (com valores arbitrários) com o número de elementos e tipo especificado, usado para otimizar desempenho quando os valores serão preenchidos logo em seguida
<code>np.array([...])</code>	Cria um array NumPy a partir de uma lista ou tupla de dados. No código, é usado tanto para listas de valores quanto para construção de um novo registro
<code>np.random.choice(array)</code>	Escolhe aleatoriamente um elemento de um array. Pode incluir o parâmetro <code>p=...</code> para atribuir probabilidades aos elementos
<code>np.datetime64(data, unidade)</code>	Converte um valor de tempo (como um <code>datetime</code>) para o tipo <code>datetime64</code> do NumPy, usado para compatibilidade com arrays estruturados e operações vetoriais
<code>np.timedelta64(valor, unidade)</code>	Representa uma diferença de tempo no formato do NumPy, usado para realizar somas com datas em arrays
<code>array['campo']</code>	Acessa os valores de um campo específico em um array estruturado. Por exemplo, <code>registros['ip']</code> retorna todos os IPs registrados
<code>np.unique(array, return_counts=True)</code>	Retorna os elementos únicos do array e, se <code>return_counts=True</code> , também a contagem de cada elemento. No código, é usado para contar falhas por IP
<code>np.append(array, novo_elemento)</code>	Retorna um novo array com o elemento adicionado ao final. No código, é usado para inserir um novo log no array de registros
<code>.astype(tipo)</code>	Converte o tipo de dados de um array ou campo. No código, é utilizado para converter <code>datetime64</code> do NumPy para objetos <code>datetime</code> do Python padrão, a fim de imprimir de forma legível

Esse código foi elaborado com o objetivo de simular, manipular e analisar registros de acesso por meio do uso de arrays estruturados com a biblioteca NumPy. Para um desenvolvedor iniciante em vetores, é importante entender como arrays podem armazenar não apenas números, mas estruturas mais complexas, como registros compostos por múltiplos campos com tipos diferentes.

O primeiro passo do código consiste na definição de um tipo de dado personalizado, chamado `dtipo`, que representa um registro de acesso. Cada registro contém quatro informações: o momento do acesso (`timestamp`), o endereço IP do cliente (`ip`), o nome do usuário (`usuario`) e o resultado da tentativa de acesso (`status`). Esses campos são definidos com tipos apropriados para o que armazenam: datas no formato de alta precisão (`datetime64[ns]`) e strings com limites de tamanho definidos (U15, U20 etc.), o que garante desempenho e uso eficiente de memória.

Em seguida, a função `gerar_logs_acesso_array` cria um array vazio de tamanho `qtd_logs` com esse tipo de dado. Após isso, começa o preenchimento de cada linha do array usando um loop. A cada iteração, o código escolhe aleatoriamente um IP, um nome de usuário e um status (com maior chance de gerar sucesso, segundo os pesos definidos). O timestamp de cada registro é incrementado progressivamente com base em um valor fixo de segundos (`intervalo_segundos`). Esse uso de `np.empty` combinado com preenchimento sequencial mostra como arrays podem ser manipulados diretamente sem criar listas intermediárias, favorecendo a performance.

A detecção de IPs suspeitos, ou seja, aqueles com um número mínimo de falhas, é feita pela função `detectar_ips_suspeitos_array`. Essa função explora operações vetoriais do NumPy, que substituem loops por comparações em massa. O filtro de falhas é feito por uma máscara booleana (`falhas_mask`), que identifica quais registros têm status igual à "falha". Em seguida, extrai-se o campo IP desses registros e calcula-se a frequência de cada IP com falhas usando `np.unique` com `return_counts=True`. Os IPs que aparecem um número de vezes maior ou igual ao limite (`limiar_falhas`) são retornados. Essa abordagem evita a repetição de lógica com `for`, mostrando a força dos vetores para análises estatísticas rápidas.

A função `inserir_log_array` trata da adição de um novo registro a esse array estruturado. Como arrays do NumPy têm tamanho fixo, a operação de inserção é feita criando-se um novo array contendo o novo registro e, em seguida, concatenando-o ao array original com `np.append`. Antes disso, é necessário converter o dicionário `novo_log` em um registro do tipo apropriado, usando `np.array(..., dtype=dtipo)`. Essa conversão garante que o novo item tenha exatamente a mesma estrutura que os demais, o que é uma exigência do NumPy para arrays homogêneos.

A função `principal`, por sua vez, organiza toda a execução. Ela gera os registros de acesso, imprime os primeiros cinco para uma pré-visualização, identifica IPs suspeitos com base em falhas repetidas, insere um novo log de falha após o último timestamp e exibe o novo total de registros. Durante a impressão, há uma conversão do timestamp de volta para objeto `datetime` do Python, facilitando a leitura. Ao final, verifica-se que o array teve seu tamanho aumentado corretamente e que o novo registro foi adicionado com sucesso.

Com esse código, um desenvolvedor iniciante em vetores pode compreender como estruturas de dados tipadas e homogêneas como arrays do NumPy podem ser utilizadas para representar tabelas, aplicar filtros lógicos, realizar contagens eficientes e até mesmo simular processos temporais. Além disso, aprende-se a lidar com operações vetoriais e conversões entre formatos de tempo, o que é fundamental para aplicações mais robustas em ciência de dados e análise de logs.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- A substituição de `np.append`, que internamente cria uma cópia do array original a cada chamada, pode ser ineficiente caso inserções sejam frequentes. Em vez disso, pode-se usar listas para inserções dinâmicas e somente ao final convertê-las em um array estruturado do tipo desejado. Caso a operação de inserção seja rara e pontual, como no exemplo fornecido, essa otimização não é essencial, mas pode ser relevante em contextos mais exigentes em termos de desempenho.

- O código também poderia se beneficiar da parametrização dos valores internos hoje fixados, como os nomes de usuários, os IPs e os pesos dos status. Essas informações poderiam ser passadas como parâmetros para a função `gerar_logs_acesso_array`, de modo que o comportamento do gerador fosse facilmente ajustável sem a necessidade de modificar o corpo da função.
- Na geração dos registros, a atualização do tempo usando `timedelta` poderia ser substituída por `np.timedelta64`, o que manteria todo o processo no escopo vetorial do NumPy. Embora `timedelta` funcione adequadamente, o uso de `np.timedelta64` tornaria o código mais coeso e evitaria conversões desnecessárias entre tipos do Python e do NumPy, o que pode trazer pequenas vantagens de desempenho e consistência.
- Outra sugestão é modularizar a lógica de exibição de registros. A impressão dos logs no principal está acoplada diretamente à lógica de execução. Criar uma função separada para formatar e exibir um registro (ou uma lista deles) aumentaria a clareza e permitiria a reutilização em outros contextos, como testes ou exportações.
- Na detecção de IPs suspeitos, embora a abordagem com `np.unique` funcione bem, o uso de `collections.Counter` também poderia ser avaliado para cenários nos quais a prioridade seja a legibilidade do código em detrimento da performance vetorial. Ainda que o NumPy seja a escolha preferencial aqui, demonstrar abordagens alternativas pode ser didático para quem está em processo de aprendizado.
- Além disso, o nome da variável `dtipo` poderia ser renomeado para algo mais descritivo, como `estrutura_log`, o que ajudaria na compreensão imediata do seu propósito. Nomes mais explícitos também são desejáveis em outras partes do código, como `tempo_atual`, que poderia ser rebatizado como `proximo_timestamp`, já que representa o instante a ser atribuído ao próximo registro e não necessariamente o tempo corrente da execução.
- Por fim, seria interessante incluir testes simples ou assertivas dentro da função principal ou em um módulo separado, validando, por exemplo, se o número de registros após a inserção é realmente um a mais, ou se a detecção de IPs suspeitos responde corretamente ao alterar o limiar. Esse tipo de validação reforça a confiabilidade do código e incentiva boas práticas de desenvolvimento com foco em testabilidade.

O entendimento da diferença entre listas e arrays em Python constitui passo essencial para programadores em início de carreira que buscam dominar a manipulação eficiente de coleções de dados. A lista, implementada como vetor dinâmico de referências para objetos, admite heterogeneidade de tipos e redimensionamento automático, ao passo que o array do módulo `array` e o `ndarray` da biblioteca NumPy oferecem armazenamento contíguo e tipagem homogênea, aspecto fundamental para desempenho numérico. Internamente, a lista reserva blocos de memória maiores que o necessário e realoca-se de forma amortizada, estratégia que torna a inserção no final praticamente custo $O(1)$ apesar de realocações ocasionais.

O acesso posicional por índice realiza-se em tempo constante, dado que o deslocamento do ponteiro base até o elemento requer apenas aritmética de endereços. A fatiagem devolve subsequências mediante

cópia, resultando em complexidade proporcional ao comprimento da fatia. Compreensões de listas fornecem mecanismo expressivo para gerar novas coleções a partir de iteração e filtragem, preservando legibilidade e evitando laços explícitos. No caso do `ndarray`, a fatiagem devolve `view` em vez de cópia, permitindo partilha de memória e menor sobrecarga, característica valiosa para grandes volumes de dados.

A pesquisa em listas demonstra custo linear quando se utiliza o operador `in`, o método `index` ou loops explícitos, uma vez que a verificação percorre cada elemento até encontrar correspondência. Quando a coleção encontra-se ordenada, o módulo `bisect` disponibiliza busca binária em $O(\log n)$, acrescentando elementos, de modo que a ordenação se mantenha. No `ndarray`, máscaras booleanas permitem localizar subconjuntos mediante operações vetorizadas, enquanto funções como `np.where` devolvem índices de ocorrência, aproveitando instruções de baixo nível para varredura rápida.

A inserção na extremidade posterior de uma lista ocorre via `append`, operação amortizada $O(1)$ em virtude da política de realocação progressiva. A extensão mediante `extend` concatena outra sequência sem criar nova lista intermediária, diferindo da soma `+`, que produz objeto adicional. Inserções em posições arbitrárias executam-se com `insert` e implicam deslocamento de todos os elementos subsequentes, resultando em $O(n)$. Para `ndarrays`, `np.append` e `np.insert` devolvem novas instâncias porque a contiguidade de memória impede expansão interna, característica que torna recomendável pré-alocar tamanho adequado ou agrupar dados em lista para posterior conversão.

A remoção na cauda da lista através de `pop()` apresenta tempo constante, pois basta reduzir o contador de tamanho; já `pop(i)` desencadeia deslocamento dos itens posteriores, elevando a complexidade para $O(n)$. A instrução `del` admite exclusão por fatiagem, enquanto `clear` zera o tamanho, preservando a alocação, alternativa útil quando se pretende reutilizar a mesma lista. Para `ndarrays`, `np.delete`, cria cópia com elementos suprimidos, comportamento que, apesar de simples, exige alocação adicional e cópia de memória, razão pela qual ciclos frequentes de remoção devem ser evitados em cenários de alto desempenho.

O módulo `array` concentra-se em tipos primitivos, como bytes ou inteiros de tamanho fixo, provendo economia de memória considerável em comparação à lista e compatibilidade com interfaces de baixo nível que exigem buffer contíguo. Operações de inserção, pesquisa ou remoção seguem as mesmas assinaturas da lista, contudo, o ganho de desempenho aparece sobretudo em leitura sequencial ou escrita em arquivos binários.

A seleção da estrutura ideal decorre da análise do perfil de uso, quantidade de elementos, frequência de inserção ou remoção e necessidade de operações matemáticas vetorizadas. Para coleções heterogêneas ou mutações frequentes na cauda, a lista mantém-se solução conveniente. Quando o foco recai sobre grandes matrizes numéricas, o `ndarray` oferece eficiência tanto em memória quanto em velocidade computacional, desde que mutações estruturais sejam raras. Já o `array` do módulo `array` situa-se como opção intermediária, entregando compacidade sem dependências externas.

Dominar tais características permite ao desenvolvedor elaborar algoritmos mais previsíveis em termos de consumo de recursos, evitando surpresas decorrentes de complexidades ocultas.

2.2 Pilhas e filas: implementação, operações e casos de uso

A compreensão de pilhas e filas em Python adquire maior profundidade quando os conceitos deixam o plano abstrato e surgem em cenários que refletem tarefas profissionais. Ao empregar exemplos ligados à robótica e aos veículos autônomos, sistemas de atendimento e blockchain, a apresentação unifica teoria e prática, permitindo que `push`, `pop`, `enqueue`, `dequeue` e `peek` apareçam como operações vitais em rotinas de produção.

Em robótica, controladores embarcados precisam lidar com uma sucessão de leituras de sensores e comandos de atuadores. A fila prioriza o primeiro evento recebido, garantindo ordem cronológica no despacho de tarefas para motores, câmeras e unidades de navegação. Quando o veículo autônomo calcula uma rota, o algoritmo de busca armazena nós visitados em uma pilha para permitir retrocesso eficiente durante a exploração de caminhos alternativos. Tal intercâmbio entre pilha e fila sustenta o equilíbrio entre exploração e execução em tempo real, principal desafio de sistemas móveis.

Nos sistemas de atendimento, a fila organiza solicitações de clientes, sejam chamadas telefônicas, tickets de chat ou mensagens instantâneas. O atendimento passa a depender da política de enfileiramento adotada, como FIFO simples para preservar ordem de chegada ou filas de prioridade que consideram nível de urgência, status do contrato e categoria de serviço. Em contrapartida, a pilha surge no back-end, armazenando estados intermediários quando um agente transfere uma interação ou consulta bases complementares, permitindo retorno ao ponto correto sem perda de contexto.

Na blockchain, transações aguardam confirmação no mempool, estrutura que se comporta como fila de prioridades. Cada operação é enfileirada conforme taxa de mineração e tamanho em bytes, aguardando seleção por nós validadores. Ao criar contratos inteligentes, desenvolvedores costumam empregar pilhas para avaliar expressões em bytecode, prática que confere eficiência ao processador virtual. A familiaridade com `push` e `pop` passa a ser requisito para auditar vulnerabilidades, como estouro de pilha ou manipulação indevida da profundidade de chamadas. Utilizando Python, a simulação de um mempool permite explorar estratégias de ordenação por taxa, enquanto a implementação de uma máquina virtual reduzida evidencia como o empilhamento controla o fluxo de execução.



Lembrete

Em robótica, controladores embarcados precisam lidar com uma sucessão de leituras de sensores e comandos de atuadores. A fila prioriza o primeiro evento recebido, garantindo ordem cronológica no despacho de tarefas para motores, câmeras e unidades de navegação.

O domínio de pilhas e filas, apoiado nos temas mencionados, desenvolve competências analíticas e de resolução de problemas ao exigir a escolha apropriada da estrutura conforme padrão de acesso, volume de dados e requisito de latência. Tal abordagem prepara o leitor para integrar componentes de software em robôs, plataformas de atendimento em larga escala ou sistemas distribuídos de registro imutável, evidenciando que dominar a teoria somente adquire verdadeiro sentido quando aplicada em desafios concretos.



Observação

Latência refere-se ao intervalo entre o momento em que um dado entra na estrutura e o instante em que se obtém resposta ou se processa a próxima etapa. Em robótica, latência reduzida garante reações rápidas; em atendimento, evita demora perceptível ao cliente; na blockchain, minimiza o tempo necessário para que uma transação seja incluída em bloco. O controle de latência envolve escolha da estrutura, afinação de algoritmos e otimização de hardware subjacente.

Para o quinto exercício, falaremos sobre robótica e os veículos autônomos que representam uma das fronteiras mais inovadoras da tecnologia, na qual a automação e a inteligência artificial se unem para transformar a mobilidade e a interação com o ambiente.

Nesse contexto, os sensores desempenham um papel crucial, coletando dados em tempo real que possibilitam ao sistema tomar decisões rápidas e precisas. O gerenciamento dessas tarefas – ou seja, a coordenação e o processamento imediato dos dados provenientes dos sensores – é essencial para garantir a segurança, a eficiência e a agilidade das operações em ambientes dinâmicos, como os encontrados em robôs e veículos autônomos.

A capacidade de gerenciar tarefas em tempo real reflete a necessidade de organizar e priorizar a entrada contínua de informações, de modo que cada dado seja processado na ordem correta para evitar atrasos ou conflitos. Em sistemas embarcados, como os utilizados em veículos autônomos, cada milissegundo conta para a tomada de decisões que pode afetar a segurança dos ocupantes e a integridade do veículo. Dessa forma, um sistema eficiente de gerenciamento de tarefas deve ser capaz de enfileirar as solicitações de processamento dos sensores e garantir que elas sejam tratadas de forma sequencial e sem perda de dados, permitindo respostas imediatas a eventos críticos.

Além disso, o gerenciamento de tarefas em tempo real é fundamental para a manutenção de uma operação robusta e confiável. Com o aumento do número de sensores e da complexidade das interações, torna-se imprescindível implementar mecanismos que organizem essas demandas de processamento, priorizando aquelas que têm maior impacto na operação do sistema. Essa abordagem melhora a eficiência e facilita a detecção e o tratamento de falhas, garantindo que o sistema se adapte rapidamente a condições variáveis e imprevistas, características comuns em ambientes de robótica e veículos autônomos.

Exemplo de aplicação

Exemplo 5 – Robótica e os veículos autônomos

Imagine que você trabalha no desenvolvimento do sistema de controle de um veículo autônomo que depende de uma variedade de sensores – como LIDAR, câmeras e sensores ultrassônicos – para monitorar o ambiente e tomar decisões em tempo real. Cada sensor gera dados que precisam ser processados imediatamente para detectar obstáculos, calcular rotas seguras e ajustar a velocidade do veículo.

Nesse cenário, o sistema deve gerenciar uma fila de tarefas que contenha essas demandas de processamento, garantindo que cada tarefa seja executada na ordem correta (seguindo o princípio FIFO).

As regras de negócio estabelecem que, ao iniciar o sistema, uma série de tarefas predefinidas deve ser enfileirada para simular o processamento inicial dos dados dos sensores. Durante a operação, novas tarefas podem ser inseridas dinamicamente, representando dados adicionais coletados ou atualizações dos sensores.

O sistema também deve remover cada tarefa após o seu processamento, assegurando que a fila se mantenha atualizada e sem tarefas duplicadas ou desnecessárias. Essa lógica é fundamental para evitar congestionamentos e garantir que as decisões baseadas nos dados sejam tomadas de forma rápida e precisa.

Adicionalmente, o sistema deve ser capaz de priorizar tarefas com base em critérios como o tipo de sensor ou a urgência dos dados processados, embora, neste exercício, a ênfase esteja na implementação básica de uma fila para o gerenciamento em tempo real.

Essa abordagem permite que o sistema simule a coordenação de tarefas em um ambiente de robótica e veículos autônomos, em que a resposta imediata é crítica para a segurança e a eficiência operacional.



Observação

O termo LIDAR deriva de "Light Detection and Ranging", ou "Detecção e Medição por Luz", e refere-se a uma tecnologia que utiliza pulsos de laser para medir distâncias com alta precisão.

O funcionamento do LIDAR baseia-se na emissão de feixes de laser em alta frequência em várias direções. Eles atingem objetos ao redor e retornam ao sensor. Medindo o tempo que o pulso leva para ir até o objeto e voltar, o sistema calcula a distância até esse objeto com grande exatidão. A partir da coleta de milhares ou milhões de pontos por segundo, o LIDAR gera uma nuvem de pontos tridimensional que representa a forma, a posição e, em alguns casos, até a densidade dos objetos no entorno do veículo.

Essa capacidade de construir mapas tridimensionais em tempo real permite ao sistema do veículo autônomo identificar obstáculos, delimitar faixas de rodagem, reconhecer bordas de calçadas, calcular espaços livres para manobra e detectar movimentos de pedestres ou outros veículos, contribuindo diretamente para uma navegação segura e autônoma.

Neste exercício, exploraremos como a linguagem Python pode ser utilizada para implementar um sistema básico de gerenciamento de tarefas em tempo real, utilizando conceitos fundamentais como listas e operações básicas de enfileiramento. Python oferece uma estrutura de dados simples, a

lista, que pode ser manipulada com métodos nativos como `append` e `pop`, os quais são essenciais para simular uma fila que segue o princípio FIFO – na qual o primeiro elemento inserido é o primeiro a ser removido. Essa característica é vital para o processamento ordenado dos dados dos sensores em sistemas de robótica e veículos autônomos.

Além das operações básicas de inserção e remoção, utilizaremos funções para modularizar o código, garantindo que cada parte do processo – desde a geração de tarefas até seu processamento – esteja bem encapsulada e organizada. Funções como `gerar_tarefas`, `adicionar_tarefa` e `processar_tarefa` serão criadas para simular as operações realizadas pelo sistema em tempo real. Essa abordagem modular facilita a manutenção do código e permite que futuras melhorias sejam implementadas de maneira incremental e sem a necessidade de reescrever todo o sistema.

Também abordaremos a importância de simular atrasos no processamento das tarefas, utilizando o módulo `time` para representar o tempo de execução real de cada uma delas. Essa simulação é crucial para entender como o gerenciamento de tarefas em um ambiente de tempo real pode ser afetado por latências e atrasos no processamento.

Ao integrar essas técnicas – manipulação de listas, definição de funções específicas e simulação de tempo –, o exercício demonstrará como construir uma solução simples, porém eficaz, para o gerenciamento de tarefas em sistemas que exigem processamento imediato dos dados coletados pelos sensores. A seguir, consta o código-fonte da solução:

```
import time
import random

def gerar_tarefas(qtd_tarefas):
    """
    Gera uma lista de tarefas simuladas para o processamento de dados dos sensores.
    Cada tarefa é representada por um dicionário que contém um 'id', uma
    'descricao' e uma 'prioridade' associada.
    """
    tarefas = []
    sensores = ["LIDAR", "câmera", "ultrassônico"]
    prioridades = ["alta", "media", "baixa"]
    for i in range(1, qtd_tarefas + 1):
        tarefa = {
            "id": i,
            "descricao": f"Processar dados do sensor {random.choice(sensores)}",
            "prioridade": random.choice(prioridades)
        }
        tarefas.append(tarefa)
    return tarefas

def adicionar_tarefa(fila_tarefas, tarefa):
    """
    Adiciona uma nova tarefa ao final da fila de tarefas.
    """
    fila_tarefas.append(tarefa)
    return fila_tarefas
```

```
def processar_tarefa(fila_tarefas):
    """
    Processa a tarefa que está no início da fila, simulando o processamento em
    tempo real.
    Remove a tarefa processada da fila e retorna a fila atualizada.
    """
    if len(fila_tarefas) > 0:
        tarefa = fila_tarefas.pop(0)
        print(f"Processando tarefa ID: {tarefa['id']}, Descrição: {tarefa['descricao']}, Prioridade: {tarefa['prioridade']}")
        time.sleep(1) # Simula o tempo de processamento da tarefa
    else:
        print("Nenhuma tarefa para processar.")
    return fila_tarefas

def principal():
    # Gerar uma fila inicial de tarefas simuladas
    qtd_inicial_tarefas = 5
    fila_tarefas = gerar_tarefas(qtd_inicial_tarefas)
    print("Fila inicial de tarefas:")
    for tarefa in fila_tarefas:
        print(tarefa)

    print("\nProcessamento das tarefas em tempo real:")
    # Processar todas as tarefas na fila
    while fila_tarefas:
        fila_tarefas = processar_tarefa(fila_tarefas)

    # Inserir uma nova tarefa simulando a chegada de dados de sensores em tempo real
    nova_tarefa = {
        "id": qtd_inicial_tarefas + 1,
        "descricao": "Processar dados do sensor GPS",
        "prioridade": "alta"
    }
    fila_tarefas = adicionar_tarefa(fila_tarefas, nova_tarefa)
    print("\nNova tarefa adicionada. Fila atual:")
    print(fila_tarefas)

if __name__ == "__main__":
    principal()
```

O quadro 5 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 5 – Funções usadas no código-fonte do quinto exercício prático

Função	Explicação de uso
time.sleep(segundos)	Interrompe a execução do programa por um número de segundos especificado. No código, simula o tempo necessário para processar uma tarefa
list.pop(índice)	Remove e retorna o elemento da lista na posição indicada. Quando usado com o índice 0, remove o primeiro elemento, simulando uma fila (estrutura FIFO)

O código inicia com a importação dos módulos `time` e `random`, essenciais para simular tanto o tempo de processamento das tarefas quanto a variabilidade dos dados dos sensores. A função `gerar_tarefas` é responsável por criar uma lista de tarefas, na qual cada uma delas é representada por um dicionário que contém um identificador, uma descrição detalhada e uma prioridade. Essa função utiliza um loop para gerar o número de tarefas especificado, escolhendo aleatoriamente o tipo de sensor e a prioridade de cada tarefa, o que reflete a diversidade de dados que um sistema de veículos autônomos pode encontrar.

A função `adicionar_tarefa` permite a inclusão de uma nova tarefa ao final da fila, mantendo a ordem de processamento conforme o princípio FIFO, fundamental para o gerenciamento em tempo real. Já a função `processar_tarefa` remove a primeira tarefa da fila utilizando o método `pop(0)` e simula o processamento dessa tarefa com uma pausa de um segundo, representando o tempo necessário para processar os dados de um sensor. Essa abordagem demonstra como as operações básicas de listas em Python podem ser utilizadas para gerenciar um fluxo contínuo de tarefas, assegurando que cada uma delas seja processada na ordem correta e que a fila seja atualizada dinamicamente à medida que novas tarefas são inseridas ou completadas.

A função principal orquestra todo o fluxo do exercício, começando pela geração de uma fila inicial de tarefas e exibindo essa fila para verificação. Em seguida, o código entra em um loop que processa cada tarefa em tempo real, removendo-as uma a uma da fila. Após o processamento inicial, uma nova tarefa é adicionada à fila para simular a chegada de dados de sensores em tempo real, e a fila atualizada é exibida. Essa sequência de operações integra os conceitos abordados na introdução teórica, mostrando de maneira prática como utilizar listas para o gerenciamento dinâmico de tarefas em um ambiente de robótica e veículos autônomos.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Uma melhoria interessante seria a utilização do módulo `queue` da biblioteca padrão do Python, que fornece uma implementação de fila thread-safe e mais robusta para o gerenciamento de tarefas em ambientes concorrentes.
- Outra possibilidade seria integrar um sistema de priorização mais avançado, no qual as tarefas com prioridades mais altas sejam processadas antes das demais, utilizando estruturas de dados como heaps ou filas de prioridade.
- Além disso, a implementação de um mecanismo de processamento assíncrono, por meio do módulo `asyncio`, poderia melhorar significativamente a eficiência e a capacidade de resposta do sistema em tempo real.
- Por fim, a criação de uma interface gráfica ou um dashboard para monitorar visualmente o status das tarefas, utilizando frameworks como Flask ou bibliotecas de visualização de dados como Plotly, permitiria uma interação mais intuitiva e o acompanhamento em tempo real das operações, ampliando o potencial da solução para aplicações reais em robótica e veículos autônomos.

Para o sexto exercício, falaremos sobre sistemas de atendimento fundamentais em diversos setores, desde call centers até serviços de suporte técnico e plataformas de e-commerce.

Esses sistemas têm a missão de gerenciar as requisições dos clientes, organizando e priorizando as solicitações para que sejam atendidas de forma ordenada e eficiente. Em um ambiente cada vez mais conectado, no qual os consumidores esperam respostas rápidas e precisas, a implementação de filas se torna essencial para manter a organização e a eficácia do atendimento.

No contexto atual, a crescente demanda por serviços automatizados e o volume de requisições geradas diariamente fazem com que os sistemas de atendimento precisem se adaptar e escalar. A implementação de filas permite que as solicitações sejam processadas sequencialmente, garantindo que cada evento ou requisição seja atendido na ordem de chegada. Tal abordagem evita a sobrecarga dos servidores e operadores humanos e contribui para a melhoria da experiência do cliente, que passa a ter um atendimento mais justo e organizado.

Além disso, a análise e o gerenciamento eficaz das filas podem proporcionar insights valiosos sobre o desempenho do sistema, permitindo a identificação de gargalos e a otimização dos processos. Ao implementar uma estrutura de fila para gerenciamento de requisições, é possível distribuir melhor os recursos e antecipar a necessidade de ajustes operacionais, fatores críticos para manter a qualidade do serviço em um mercado competitivo. Essa prática se torna ainda mais relevante à medida que as organizações buscam automatizar e agilizar o atendimento, garantindo respostas rápidas e precisas em um ambiente de alta demanda.

Neste exercício, exploraremos os recursos nativos do Python para implementar uma fila que gerencia requisições de atendimento de forma sequencial e eficiente. Utilizaremos a estrutura de dados lista, que é extremamente versátil em Python, permitindo armazenar e manipular coleções de dados de maneira dinâmica. A operação de enfileiramento será realizada por meio dos métodos `append` para adicionar novos elementos no final da lista e `pop(0)` para remover e retornar o primeiro elemento da lista, garantindo o processamento na ordem de chegada dos eventos. Essa abordagem é simples, mas muito eficaz para simular o comportamento de um sistema de atendimento.

Outra característica importante abordada será a modularização do código através da definição de funções específicas para cada operação: geração de requisições simuladas, inserção de novas requisições e processamento de eventos. Essa técnica facilita a manutenção do código e a implementação de futuras melhorias, além de promover a clareza e a organização do programa. Ao dividir o sistema em funções bem definidas, o exercício demonstra como isolar responsabilidades em pequenos blocos de código, tornando o sistema mais legível e de fácil compreensão.

Adicionalmente, enfatizaremos a importância da manipulação de dados em tempo real e da simulação de atrasos de processamento, utilizando o módulo `time`. Esse recurso permite simular o tempo gasto no processamento de cada requisição, aproximando a implementação de um cenário real quando o tempo de resposta pode variar de acordo com a complexidade da tarefa.

A integração entre a manipulação de listas, o uso de funções e a simulação de tempo real forma a base para entender como um sistema de atendimento pode ser estruturado em Python, preparando

o leitor para desenvolver aplicações mais sofisticadas e adaptáveis a ambientes de alta demanda. A seguir, consta o código-fonte da solução:

```
import time

def gerar_requisicoes(qtd_requisicoes):
    """
    Gera uma lista de requisições simuladas para um sistema de atendimento.
    Cada requisição é representada por um dicionário contendo:
    - 'id': identificador único da requisição,
    - 'cliente': nome do cliente,
    - 'problema': descrição do problema relatado,
    - 'horario': horário de chegada da requisição.
    """
    requisicoes = []
    nomes_clientes = ["Cliente_A", "Cliente_B", "Cliente_C", "Cliente_D"]
    problemas = ["Problema de conexão", "Erro no sistema", "Dúvida sobre
faturamento", "Solicitação de suporte"]
    for i in range(1, qtd_requisicoes + 1):
        requisicao = {
            "id": i,
            "cliente": nomes_clientes[i % len(nomes_clientes)],
            "problema": problemas[i % len(problemas)],
            "horario": f"{time.strftime('%H:%M:%S')}"
        }
        requisicoes.append(requisicao)
        time.sleep(0.1) # Simula um intervalo entre as requisições
    return requisicoes

def adicionar_requisicao(fila_requisicoes, nova_requisicao):
    """
    Adiciona uma nova requisição ao final da fila de atendimento.
    """
    fila_requisicoes.append(nova_requisicao)
    return fila_requisicoes

def processar_requisicao(fila_requisicoes):
    """
    Processa a requisição que está no início da fila.
    Remove a requisição processada e retorna a fila atualizada.
    """
    if fila_requisicoes:
        requisicao = fila_requisicoes.pop(0)
        print(f"Processando requisição ID: {requisicao['id']} | Cliente:
{requisicao['cliente']} | Problema: {requisicao['problema']} | Horário:
{requisicao['horario']}")
        time.sleep(1) # Simula o tempo de processamento da requisição
    else:
        print("Nenhuma requisição para processar.")
    return fila_requisicoes

def principal():
    # Geração de uma fila inicial de requisições simuladas
    qtd_inicial_requisicoes = 5
    fila_requisicoes = gerar_requisicoes(qtd_inicial_requisicoes)
```

```
print("Fila inicial de requisições:")
for requisicao in fila_requisicoes:
    print(requisicao)

print("\nProcessamento das requisições em ordem:")
# Processamento das requisições em ordem FIFO
while fila_requisicoes:
    fila_requisicoes = processar_requisicao(fila_requisicoes)

# Simulando a chegada de uma nova requisição
nova_requisicao = {
    "id": qtd_inicial_requisicoes + 1,
    "cliente": "Cliente_E",
    "problema": "Reclamação de atraso no serviço",
    "horario": f"{time.strftime('%H:%M:%S')}"
}
fila_requisicoes = adicionar_requisicao(fila_requisicoes, nova_requisicao)
print("\nNova requisição adicionada. Fila atual:")
print(fila_requisicoes)

if __name__ == "__main__":
    principal()
```

O quadro 6 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 6 – Funções usadas no código-fonte do sexto exercício prático

Função	Explicação de uso
<code>time.strftime('%H:%M:%S')</code>	Retorna a hora atual formatada como uma string no formato horas:minutos:segundos. No código, é utilizada para registrar o horário de chegada das requisições
<code>i % len(lista)</code>	Calcula o resto da divisão de <code>i</code> pelo comprimento da lista. Essa operação é usada para percorrer elementos de forma cíclica, garantindo que os índices se mantenham dentro dos limites da lista

O código inicia importando o módulo `time`, que é utilizado tanto para simular intervalos entre a geração das requisições quanto para representar o tempo de processamento de cada evento, aproximando a simulação de um ambiente real de atendimento. A função `gerar_requisicoes` é responsável por criar uma lista de requisições, na qual cada requisição é modelada como um dicionário contendo um identificador único, o nome do cliente, a descrição do problema e o horário em que a requisição foi registrada. Ao iterar um número definido de vezes, a função utiliza `time.strftime` para capturar o horário atual e adiciona um pequeno atraso com `time.sleep(0.1)`, simulando a chegada escalonada de requisições.

A função `adicionar_requisicao` serve para inserir uma nova requisição ao final da fila, utilizando o método `append` da lista, o que é fundamental para manter a ordem de chegada dos eventos. Por outro lado, a função `processar_requisicao` implementa o processamento sequencial dos eventos utilizando o método `pop(0)` para remover e retornar o primeiro elemento da fila, garantindo que a requisição mais antiga seja atendida primeiro, conforme o princípio FIFO. Durante o processamento, a função exibe os detalhes da requisição e simula um tempo de atendimento com `time.sleep(1)`, que representa o tempo gasto para processar a solicitação.

A função principal integra todas as operações, iniciando com a geração de uma fila de requisições simuladas e exibindo seu conteúdo para verificação. Em seguida, um loop é executado para processar todas as requisições existentes na fila, removendo-as uma a uma até que a fila esteja vazia. Após esse processamento, uma nova requisição é adicionada à fila para simular a chegada contínua de novos eventos e a fila atualizada é exibida. Esse fluxo demonstra a aplicação dos conceitos de manipulação de listas, operações de enfileiramento e modularização do código, evidenciando como um sistema de atendimento pode ser implementado de forma simples e eficiente em Python.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Explorar o uso do módulo `queue` da biblioteca padrão do Python, que oferece uma implementação de filas `thread-safe`, ideal para ambientes concorrentes e sistemas que exigem alta confiabilidade.
- A implementação de um mecanismo de priorização de requisições, utilizando filas de prioridade ou estruturas de dados como `heaps`, pode permitir que eventos mais críticos sejam processados antes dos demais, agregando uma camada de inteligência ao sistema.
- Outra melhoria viável seria a integração com frameworks de processamento assíncrono, como o `asyncio`, para possibilitar o processamento paralelo de múltiplas requisições e reduzir a latência geral do atendimento.
- Finalmente, o desenvolvimento de uma interface gráfica ou dashboard para monitoramento em tempo real do status das requisições, utilizando frameworks web como `Flask` ou `Django` e bibliotecas de visualização como `Plotly`, pode transformar a solução em uma ferramenta interativa e mais robusta para ambientes de atendimento de alta demanda.

Para o sexto exercício, falaremos sobre a tecnologia `blockchain` que revolucionou a forma como transacionamos e armazenamos informações, trazendo uma nova era para a segurança digital e para a execução de contratos inteligentes – os famosos `smart contracts`.

Em sua essência, o `blockchain` é um sistema distribuído e descentralizado que registra transações em blocos interligados, garantindo transparência, imutabilidade e segurança. Essa infraestrutura tem ganhado destaque em diversos setores, desde finanças até cadeias de suprimentos, pois permite a verificação confiável de operações sem a necessidade de intermediários centralizados.

No contexto dos `smart contracts`, as operações de transação são executadas de maneira autônoma e, muitas vezes, precisam de mecanismos para reverter ou ajustar ações quando ocorrem erros ou mudanças nas condições de mercado. O uso de pilhas – estrutura de dados que segue o princípio `LIFO` – é particularmente adequado para controlar operações temporárias, como `undo` (desfazer) e `redo` (refazer) de transações. Essa abordagem permite que, caso uma operação seja executada de forma indevida ou ocorra algum problema, o sistema possa reverter as ações realizadas na ordem inversa à que foram aplicadas, mantendo a integridade e a consistência do `smart contract`.

A implementação de pilhas para gerenciar transações temporárias em `smart contracts` é crucial para oferecer flexibilidade e segurança nos processos de execução. Em um ambiente dinâmico em que as transações

podem ser complexas e interdependentes, ter um mecanismo de reversão ajuda a mitigar riscos e corrigir falhas sem comprometer todo o sistema. Esse gerenciamento de operações é especialmente importante para aplicações financeiras e outros domínios sensíveis, nos quais cada transação pode ter implicações significativas e a capacidade de desfazer ou refazer operações pode evitar prejuízos e aumentar a confiabilidade do sistema.

Exemplo de aplicação

Exemplo 6 – Blockchain

Imagine que você faz parte da equipe de desenvolvimento de uma plataforma descentralizada que utiliza smart contracts para gerenciar operações financeiras automatizadas. Nesta plataforma, cada transação – como transferência de ativos ou execução de ordens – é registrada em uma pilha, permitindo que, caso alguma operação seja realizada erroneamente, seja possível utilizar um mecanismo de undo para desfazer a transação, ou redo para refazê-la, se necessário.

As regras de negócio definem que toda transação deve ser armazenada na pilha principal e, ao ser desfeita, movida para uma pilha de transações desfeitas.

Se uma nova transação for inserida, a pilha de redo é limpa para evitar inconsistências. Essa lógica garante que as operações sejam revertidas na ordem correta e que o sistema possa restaurar o estado anterior, conforme as necessidades de auditoria e correção de erros.

Neste exercício, exploraremos os fundamentos de manipulação de estruturas de dados em Python, com foco na implementação de pilhas para controlar operações de transação temporárias e reversões em smart contracts. Python oferece recursos nativos para trabalhar com listas, que podem ser utilizadas para simular o comportamento de uma pilha. Utilizando métodos como `append()` para empurrar (push) elementos para o topo da pilha e `pop()` para removê-los na ordem inversa, conseguimos implementar o princípio LIFO de forma simples e eficiente. Essa abordagem permite que as transações sejam armazenadas e revertidas conforme necessário, refletindo a lógica utilizada em sistemas de blockchain.



Saiba mais

Para se aprofundar na tecnologia blockchain, a obra a seguir apresenta exemplos de criação de chaves, endereços multisignature e APIs de cadeia de blocos; permitindo que o leitor programe transações e experimente mineração nos primeiros exercícios, característica que favorece quem precisa incluir rotinas de Bitcoin em scripts Python de modo imediato.

GARG, H. K. *Hands-on bitcoin programming with Python: build powerful online payment centric applications with Python*. Birmingham: Packt Publishing, 2018.

Já o seguinte livro oferece tratamento abrangente do ecossistema Ethereum ao combinar contratos inteligentes em Vyper, integração via web3.py, construção de tokens e carteiras gráficas, além de demonstrar o armazenamento descentralizado com IPFS. Tal conjunto cobre todas as etapas exigidas para colocar em produção uma aplicação distribuída escrita em Python.

KOK, A. S. *Hands-on blockchain for Python developers: gain blockchain programming skills to build decentralized applications using Python*. Birmingham: Packt Publishing, 2019.

De modo complementar, na obra a seguir, o leitor observa a construção de uma blockchain funcional a partir do zero, aprofundando prova de trabalho, criptografia aplicada e redes ponto a ponto, de modo que cada capítulo associa teoria algorítmica ao desenvolvimento incremental do código em Python, consolidando a compreensão estrutural de todo o sistema.

VAN FLYMEN, D. *Learn blockchain by building one: a concise path to understanding cryptocurrencies*. Berkeley: Apress, 2020.

Além disso, o exercício fará uso da modularização do código através da definição de funções específicas para cada operação: adicionar uma transação, desfazer (undo) uma transação e refazer (redo) uma transação. Essa estratégia de dividir o problema em partes menores não apenas torna o código mais organizado e legível, como facilita a manutenção e a extensão da funcionalidade. A modularização é fundamental em aplicações reais, em que diferentes partes do sistema podem ser atualizadas ou melhoradas sem a necessidade de reescrever todo o código. Dessa forma, o exercício demonstra como aplicar conceitos básicos de programação para resolver problemas complexos de gerenciamento de transações.

Por fim, enfatizaremos a importância de manter o estado do sistema consistente, utilizando duas pilhas distintas – uma para as transações realizadas e outra para as transações desfeitas. Essa separação é essencial para garantir que, ao desfazer uma operação, seja possível também refazê-la se necessário, mantendo um registro claro do histórico de operações. A integração desses conceitos com o uso de funções nativas de Python proporciona uma base sólida para a construção de sistemas robustos, como os smart contracts, que exigem mecanismos eficientes de reversão de operações para assegurar a integridade e a confiabilidade dos processos transacionais. A seguir, consta o código-fonte da solução:

```
def adicionar_transacao(pilha_transacoes, transacao):  
    """  
    Adiciona uma nova transação à pilha principal.  
    """  
    pilha_transacoes.append(transacao)  
    return pilha_transacoes  
  
def desfazer_transacao(pilha_transacoes, pilha_transacoes_desfeitas):  
    """  
    Remove a transação do topo da pilha principal e a adiciona à pilha de  
    transações desfeitas.
```

```
"""
if pilha_transacoes:
    transacao = pilha_transacoes.pop()
    pilha_transacoes_desfeitas.append(transacao)
    print(f"Transação desfeita: {transacao}")
else:
    print("Nenhuma transação para desfazer.")
return pilha_transacoes, pilha_transacoes_desfeitas

def refazer_transacao(pilha_transacoes, pilha_transacoes_desfeitas):
    """
    Remove a transação do topo da pilha de transações desfeitas e a retorna para
    a pilha principal.
    """
    if pilha_transacoes_desfeitas:
        transacao = pilha_transacoes_desfeitas.pop()
        pilha_transacoes.append(transacao)
        print(f"Transação refeita: {transacao}")
    else:
        print("Nenhuma transação para refazer.")
    return pilha_transacoes, pilha_transacoes_desfeitas

def exibir_pilhas(pilha_transacoes, pilha_transacoes_desfeitas):
    """
    Exibe o conteúdo das pilhas de transações e de transações desfeitas.
    """
    print("\nPilha de Transações Atuais:")
    for t in pilha_transacoes:
        print(t)
    print("\nPilha de Transações Desfeitas:")
    for t in pilha_transacoes_desfeitas:
        print(t)

def principal():
    # Pilhas para armazenar transações
    pilha_transacoes = []
    pilha_transacoes_desfeitas = []

    # Adicionando transações iniciais
    transacoes_iniciais = [
        "Transação 1: Envio de 100 tokens",
        "Transação 2: Recebimento de 50 tokens",
        "Transação 3: Pagamento de 30 tokens",
        "Transação 4: Envio de 20 tokens"
    ]

    for transacao in transacoes_iniciais:
        pilha_transacoes = adicionar_transacao(pilha_transacoes, transacao)

    print("Estado inicial das pilhas:")
    exibir_pilhas(pilha_transacoes, pilha_transacoes_desfeitas)

    # Desfazendo duas transações
    pilha_transacoes, pilha_transacoes_desfeitas = desfazer_transacao(pilha_
transacoes, pilha_transacoes_desfeitas)
    pilha_transacoes, pilha_transacoes_desfeitas = desfazer_transacao(pilha_
```

```
transacoes, pilha_transacoes_desfeitas)

print("\nApós desfazer duas transações:")
exibir_pilhas(pilha_transacoes, pilha_transacoes_desfeitas)

# Refazendo uma transação
pilha_transacoes, pilha_transacoes_desfeitas = refazer_transacao(pilha_
transacoes, pilha_transacoes_desfeitas)

print("\nApós refazer uma transação:")
exibir_pilhas(pilha_transacoes, pilha_transacoes_desfeitas)

if __name__ == "__main__":
    principal()
```

O quadro 7 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições. As funções já explicitadas nos quadros anteriores foram omitidas para evitar duplicidade.

Quadro 7 – Funções usadas no código-fonte do sexto exercício prático

Função	Explicação de uso
list como pilha (append + pop sem índice)	A lista é utilizada como estrutura de pilha (LIFO), com <code>append()</code> para empilhar e <code>pop()</code> (sem argumento) para desempilhar o último elemento inserido
for elemento in lista	Itera sobre todos os elementos de uma lista, executando um bloco de código para cada item. No código, é usado para percorrer as transações e exibi-las na tela

O código inicia definindo funções específicas para manipular as pilhas de transações, utilizando a estrutura de dados lista para simular o comportamento de uma pilha, que opera segundo o princípio LIFO. A função responsável por adicionar uma nova transação, denominada `adicionar_transacao`, utiliza o método `append()` para empurrar o novo registro para o topo da pilha principal. Essa operação é essencial para registrar cada operação realizada pelo smart contract, mantendo um histórico ordenado das transações efetuadas.

Em seguida, a função `desfazer_transacao` implementa o mecanismo de undo, removendo a transação do topo da pilha principal com o método `pop()` e, em seguida, adicionando essa transação à pilha de transações desfeitas. Essa abordagem permite que o sistema reverta a operação mais recente, facilitando a correção de erros ou a necessidade de reverter uma ação sem alterar a ordem dos registros anteriores. A função `refazer_transacao`, por sua vez, realiza o processo inverso, retirando a transação do topo da pilha de transações desfeitas e retornando-a para a pilha principal, possibilitando o redo da operação desfeita.

A função principal integra todas as funcionalidades, iniciando com a criação de uma pilha de transações com um conjunto de operações simuladas. O fluxo do programa demonstra o estado inicial das pilhas, seguido pela execução de dois comandos de desfazer (undo) que movem as transações mais recentes para a pilha de transações desfeitas. Em seguida, uma operação de refazer (redo) é aplicada, reintegrando uma das transações desfeitas à pilha principal. O uso de funções específicas para cada operação, aliado à exibição do estado das pilhas, ilustra como o gerenciamento de transações temporárias pode ser realizado de forma

clara e modular, reforçando os conceitos fundamentais da estrutura de dados pilha e sua aplicação em smart contracts para operações de undo/redo.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Integrar um mecanismo de priorização que permita o gerenciamento de transações com diferentes níveis de importância, possivelmente utilizando uma estrutura de dados de fila de prioridade com a pilha.
- Outra melhoria relevante seria a implementação de tratamento de exceções, garantindo que operações de undo ou redo sejam realizadas de forma segura mesmo em casos de pilhas vazias ou inconsistências no histórico de transações.
- A incorporação de uma interface gráfica simples ou até mesmo uma API RESTful para monitoramento das transações pode transformar a solução em uma ferramenta interativa, facilitando a visualização e o controle das operações em tempo real.
- Ademais, a integração com bibliotecas específicas para smart contracts, como frameworks de blockchain, permitiria uma simulação mais realista, em que as operações de undo/redo poderiam ser testadas em um ambiente de desenvolvimento descentralizado, aproximando ainda mais a implementação dos desafios enfrentados no mundo real.

A compreensão de pilhas e filas representa etapa fundamental na formação de quem inicia a trajetória como desenvolvedor, pois ambas estruturam o fluxo de dados de maneira disciplinada e previsível. Na pilha, a política LIFO determina que o elemento mais recente seja sempre o próximo a ser removido, característica obtida por meio de inserções e remoções no mesmo extremo. Já a fila adota abordagem FIFO, na qual o primeiro item inserido torna-se o primeiro a sair, fornecendo ordem natural de atendimento semelhante a linhas de espera em ambientes físicos.

Em Python, a pilha pode ser construída diretamente com `list`, cuja função `append` adiciona elementos no fim em tempo amortizado $O(1)$ graças ao regime de realocação progressiva do vetor dinâmico, enquanto `pop()` remove a última referência com custo equivalente. A ausência de sincronização interna impede uso seguro em múltiplas threads sem bloqueios adicionais, porém fornece desempenho elevado em cenários de execução isolada. A fila implementada com `list` por meio de `pop(0)` sofre penalidade $O(n)$ devido ao deslocamento dos itens restantes, razão pela qual essa alternativa não se recomenda para largas sequências.

A classe `deque` no pacote `collections` oferece acesso eficiente em ambos os extremos, já que é baseada em blocos duplamente encadeados que eliminam cópia de memória durante inserções ou remoções frontais. A operação `appendleft` viabiliza construção de pilhas reversas, enquanto `popleft` realiza remoções de fila em $O(1)$ sem o custo de realocar a estrutura. Por rejeitar bloqueios, o `deque` preserva desempenho em ambiente `monothread`, contudo requer precaução caso seja

compartilhado entre threads sem mecanismos de exclusão mútua. Reescreveremos o código do quinto exercício deste tópico, usando esses conceitos:

```
import time
import random
from collections import deque # estrutura FIFO eficiente em memória
from queue import Queue # alternativa threadsafe, usada se houver múltiplas threads

# ----- VERSÃO MONOTHREAD COM deque -----

def gerar_tarefas(qtd_tarefas: int) -> deque:
    """
    Cria uma fila de tarefas simuladas utilizando deque.
    Cada tarefa possui id, descrição e prioridade.
    """
    sensores = ["LIDAR", "câmera", "ultrassônico"]
    prioridades = ["alta", "média", "baixa"]
    fila = deque()
    for i in range(1, qtd_tarefas + 1):
        fila.append(
            {
                "id": i,
                "descricao": f"Processar dados do sensor {random.
choice(sensores)}",
                "prioridade": random.choice(prioridades),
            }
        )
    return fila

def adicionar_tarefa(fila_tarefas: deque, tarefa: dict) -> deque:
    """
    Adiciona uma nova tarefa ao fim da fila em O(1).
    """
    fila_tarefas.append(tarefa)
    return fila_tarefas

def processar_tarefa(fila_tarefas: deque) -> deque:
    """
    Remove e processa a tarefa que está no início da fila em O(1).
    """
    if fila_tarefas:
        tarefa = fila_tarefas.popleft()
        print(
            f"Processando tarefa ID: {tarefa['id']}, "
            f"Descrição: {tarefa['descricao']}, "
            f"Prioridade: {tarefa['prioridade']}"
        )
        time.sleep(1) # simula tempo de processamento
    else:
        print("Nenhuma tarefa para processar.")
    return fila_tarefas

def principal():
    qtd_inicial = 5
```

```
fila = gerar_tarefas(qtd_inicial)

print("Fila inicial de tarefas:")
for t in fila:
    print(t)

print("\nProcessamento das tarefas em tempo real:")
while fila:
    processar_tarefa(fila)

nova_tarefa = {
    "id": qtd_inicial + 1,
    "descricao": "Processar dados do sensor GPS",
    "prioridade": "alta",
}
adicionar_tarefa(fila, nova_tarefa)

print("\nNova tarefa adicionada. Fila atual:")
print(list(fila)) # converte deque para lista apenas para exibir

# ----- ALTERNATIVA THREADSAFE COM queue.Queue -----

def gerar_fila_threadsafe(qtd_tarefas: int) -> Queue:
    """
    Cria uma fila FIFO thread-safe que permite múltiplos produtores e
    consumidores.
    """
    sensores = ["LIDAR", "câmera", "ultrassônico"]
    prioridades = ["alta", "média", "baixa"]
    fila_segura = Queue()

    for i in range(1, qtd_tarefas + 1):
        fila_segura.put(
            {
                "id": i,
                "descricao": f"Processar dados do sensor {random.
choice(sensores)}",
                "prioridade": random.choice(prioridades),
            }
        )
    return fila_segura

if __name__ == "__main__":
    principal()
```

O programa original tratava a fila de tarefas com listas comuns, procedimento funcional, porém pouco eficiente quando o número de elementos cresce, porque cada remoção frontal reposiciona todos os demais itens internamente. A nova implementação substitui a lista por deque, estrutura que mantém ponteiros duplamente encadeados e viabiliza operações de inserção no final e remoção no início em tempo constante, preservando a ordem de chegada e eliminando cópias de memória. A chamada `popleft` retira a tarefa mais antiga sem deslocar os elementos restantes, enquanto `append` mantém a mesma assinatura utilizada anteriormente, fato que reduz o impacto na leitura do código por quem já conhecia a versão original.

A inclusão das anotações de tipo oferece ao leitor ideia clara do que cada função espera e devolve, contribuindo para depuração e manutenção. Quando o cenário exigir comunicação entre várias threads, o deque não fornece garantia de exclusão mútua, por essa razão surge uma alternativa baseada em `queue.Queue`. Essa classe incorpora travas internas em cada operação de enfileiramento e desenfileiramento, permitindo que produtores coloquem tarefas e consumidores as retirem sem risco de condição de corrida, bastando trocar as chamadas de criação da fila e empregar `put` e `get` no lugar de `append` e `popleft`. Com essa abordagem, a lógica central permanece, porém, a infraestrutura passa a lidar com sincronização de forma transparente ao desenvolvedor.

Ao adotar deque no ambiente `monothread` e reservar `Queue` para eventuais cenários concorrentes, o código passa a explorar recursos nativos da biblioteca padrão que foram concebidos exatamente para lidar com filas de forma eficiente e segura, evitando sobrecarga de processamento e fornecendo caminho natural para evoluir o sistema quando a demanda por paralelismo surgir.

Para aplicações concorrentes, o módulo `queue` disponibiliza instâncias `threadsafe.Queue`. `Queue` mantém disciplina FIFO, `LifoQueue` reproduz pilha e `SimpleQueue` fornece variação simplificada. Cada operação de enfileiramento ou desenfileiramento inclui bloqueio interno, permitindo troca de dados entre produtores e consumidores sem riscos de condição de corrida. Os métodos `put` e `get` podem bloquear até que haja espaço ou elementos disponíveis, integrando-se a algoritmos que precisem de coordenação temporal.

Embora as bibliotecas padrão sejam suficientes em grande parte das tarefas, a implementação manual com nós encadeados ainda possui relevância didática. Um nó contém valor e ponteiro para o próximo, permitindo crescimento contínuo sem realocação de blocos contíguos. A inserção na cabeça de uma pilha encadeada e a remoção subsequente preservam complexidade $O(1)$, porém acessos aleatórios exigem travessia da lista, custo inadmissível quando índices específicos são necessários com frequência. Na fila encadeada, a manutenção de ponteiros para cabeça e cauda garante inserção e remoção em extremos opostos, também com custo constante.

As operações essenciais em pilhas incluem `push`, `pop` e verificação de elemento no topo, frequentemente denominada `peek`. Cada uma delas realiza-se em tempo constante quando sustentada por `list` ou deque.

Outros casos de uso para pilhas abrangem avaliação de expressões aritméticas, verificação de balanço de parênteses, algoritmos de retrocesso e percurso em profundidade em grafos. O mecanismo de chamada de funções da máquina virtual Python depende de pilha para armazenar contextos de ativação, ilustrando importância prática dessa estrutura na execução de programas. Editores de texto e navegadores implementam histórico de desfazer e refazer recorrendo ao mesmo conceito, o que demonstra versatilidade além do domínio acadêmico.

Filas, por sua vez, alinham-se naturalmente à comunicação assíncrona e ao gerenciamento de recursos compartilhados. Algoritmos de busca em largura utilizam-nas para explorar níveis de um grafo de modo ordenado, enquanto servidores de impressão armazenam trabalhos em ordem de chegada. Em sistemas distribuídos, brokers de mensagens como RabbitMQ ou Kafka empregam filas para desacoplar

produtores de consumidores, mantendo vazão estável sob picos de carga. Buffers de áudio e vídeo dependem da mesma disciplina para reproduzir mídia sem interrupções.

A seleção da implementação mais adequada decorre do volume de dados, da necessidade de concorrência e do perfil de chamada das operações fundamentais. Quando o ambiente é `singlethread` e a ênfase recai em empilhar ou desempilhar rapidamente, `list` atende com simplicidade. Para filas extensas atualizadas em ambas extremidades, `deque` oferece desempenho superior. Situações envolvendo threads múltiplas devem recorrer às classes do módulo `queue` a fim de garantir integridade. Em qualquer cenário, a ferramenta `timeit` ajuda a quantificar impacto das escolhas, fornecendo evidência empírica que direciona decisões fundamentadas.

Ao dominar esses detalhes, o desenvolvedor passa a projetar algoritmos previsíveis em consumo de tempo e memória, prevenindo gargalos ocasionados por escolhas inadequadas de estrutura. A experimentação constante solidificará a intuição sobre quando empregar cada variante, favorecendo soluções mais eficientes.



Resumo

Nesta unidade, exploramos em profundidade os conceitos fundamentais sobre algoritmos e estruturas de dados, utilizando Python como a principal linguagem de programação devido à sua clareza e simplicidade. Destacamos a importância da análise de complexidade computacional, utilizando principalmente a notação Big-O para descrever o desempenho esperado de algoritmos em relação ao crescimento do tamanho da entrada. Exemplos concretos, como operações que ocorrem em tempo constante ($O(1)$) e outras em tempo linear ($O(n)$), foram apresentados para ilustrar o conceito.

Na sequência, apresentamos as estruturas de dados lineares como sequências ordenadas que ligam cada elemento ao seu predecessor e ao seu sucessor, característica que favorece a execução de operações de pesquisa, inserção e remoção em posição conhecida. Listas e arrays surgiram como formas versáteis de organizar informações em Python: listas aceitam elementos heterogêneos e oferecem métodos como `append`, `insert` e `pop`; enquanto arrays, tipados e contíguos reduzem consumo de memória e atendem a cenários de cálculo numérico intenso. Pilhas, governadas pela política LIFO, e filas, regidas por FIFO, completam esse conjunto linear, fornecendo controle rigoroso sobre a ordem de processamento de dados.

Ao longo do texto, intercalamos teoria e prática ao relacionar cada estrutura a contextos profissionais atuais, conforme quadro a seguir.

Quadro 8 – Relação entre exercícios práticos desenvolvidos neste tópico e contextos profissionais reais

Estrutura de dados	Áreas de aplicação	Exercício – Utilização específica
Listas e arrays	Big Data e IoT	Manipulação de grandes volumes de dados de sensores e dispositivos conectados
		Processamento de séries temporais e logs para análises preditivas
	Computação em nuvem	Armazenamento e manipulação de dados em bancos NoSQL e sistemas distribuídos
	Cibersegurança	Registro e análise de logs de acesso, possibilitando a detecção de atividades suspeitas
Pilhas e filas	Robótica e veículos autônomos	Gerenciamento de tarefas em tempo real, coordenando processamento de dados dos sensores
	Sistemas de atendimento	Implementação de filas para gerenciamento de requisições e processamento sequencial de eventos
	Blockchain	Uso de pilhas para controlar operações de transação temporárias ou reversões (undo/redo) em smart contracts



Exercícios

Questão 1. (Fadesp 2021, adaptada) Avalie o código-fonte a seguir, desenvolvido utilizando a linguagem Python.

```
1.  numeros = [10, 20, 30, 40, 50]
2.
3.  def calcular_md(lista):
4.      if len(lista) == 0:
5.          return 0
6.      return sum(lista) / len(lista)
7.
8.  md = calcular_md(numeros)
9.  print(f"{md:.2f}")
```

Qual é a saída gerada pelo programa?

A) 18.95

B) 20.00

C) 25.00

D) 30.00

E) 37.80

Resposta correta: alternativa D.

Análise da questão

O código começa com a declaração de uma lista `numeros`, contendo 5 elementos. Na linha 8, a função `calcular_md()` é chamada, enviando `numeros` como argumento.

A execução é desviada para a linha 3, na qual ocorre a definição da função. Ali, o parâmetro `lista` recebe o endereço de memória de `numeros`. O comando condicional da linha 4 testa se o número de elementos da lista é igual a zero. Como a lista tem 5 elementos, o comando de retorno da linha 5 não será executado.

A linha 6, no entanto, entrará em execução, retornando à variável `md` o resultado da operação `sum(lista)/len(lista)`. O comando `sum(lista)` trará o resultado do somatório dos elementos da lista. Temos que $10 + 20 + 30 + 40 + 50 = 150$. O comando `len(lista)` trará o número de elementos da lista, que é igual a 5. Logo, o retorno gerado pela função `calcular_md()` será $150/5 = 30$. Por fim, o comando da linha 9 imprimirá o valor associado à variável `md` com duas casas decimais, o que gerará a saída 30.00.

Questão 2. (Cesgranrio 2024, adaptada) A figura a seguir exibe uma fila e uma pilha de números inteiros.

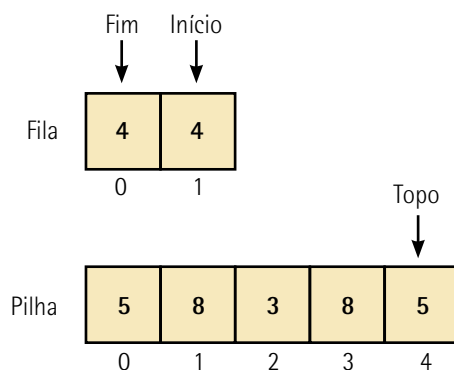


Figura 9

O código Python a seguir implementa essas estruturas de dados por meio de listas.

```
pilha = [5, 8, 3, 8, 5]
```

```
pilAux = [ ]
```

```
fila = [4, 4]
```

Admita que o módulo `pilha` contém as funções `push(pil, e)` e `pop(pil)`, que implementam as operações usuais sobre uma pilha. O módulo `fila` contém as funções `enqueue(fila, e)` e `dequeue(fila)`, que implementam as operações usuais sobre uma fila. Ambos os módulos serão importados por um programa Python. Após a definição das estruturas de dados, esse programa Python executa uma sequência de comandos, de modo que, ao término da execução, as variáveis `pilha` e `fila` referenciam listas iguais.

Qual é essa sequência de comandos?

A) `push(pilAux,pop(pilha))`

`push(pilAux,pop(pilha))`

`pop(pilha)`

`enqueue(fila,pop(pilAux))`

`enqueue(fila,pop(pilAux))`

B) `push(pilAux, pop(pilha))`

`enqueue(fila, pop(pilha))`

`enqueue(fila, pop(pilAux))`

`push(pilha, dequeue(fila))`

C) `push(pilAux, pop(pilha))`

`enqueue(fila, pop(pilha))`

`enqueue(fila, pop(pilAux))`

`pop(pilha)`

`push(pilha, dequeue(fila))`

D) `enqueue(fila, pop(pilha))`

`enqueue(fila, pop(pilha))`

`pop(pilha)`

`push(pilha, dequeue(fila))`

E) `enqueue(fila, pop(pilha))`

`enqueue(fila, pop(pilha))`

`push(pilha, dequeue(fila))`

Resposta correta: alternativa C.

Análise da questão

No contexto de estruturas de dados, os métodos `push()` e `pop()` são métodos de manipulação de pilhas. Já `enqueue()` e `dequeue()` são usados para manipular filas.

De acordo com as definições de métodos entregues pelo enunciado e pela literatura de estruturas de dados, estamos lidando com os métodos a seguir.

`push(pil, e)`: este método recebe a pilha a ser manipulada no parâmetro `pil` e um elemento no parâmetro `e`. Ele coloca o valor de `e` no último índice da pilha, que representa o seu início.

`pop(pilha)`: esse método recebe como argumento a pilha a ser manipulada, remove o elemento de topo (ou seja, aquele que está no último índice da pilha) e o retorna.

`enqueue(fila, e)`: este método recebe como argumento a fila a ser manipulada no parâmetro `fila` e um elemento no parâmetro `e`. Ele insere o valor de `e` no índice 0 da fila, que representa o seu fim.

`dequeue(fila)`: este método recebe como argumento a fila a ser manipulada, remove o elemento de índice mais alto (que representa o início da fila) e o retorna.

Temos a declaração de três listas, cuja situação inicial é reproduzida a seguir.

```
pilha = [5, 8, 3, 8, 5]
```

```
pilhaAux = [ ]
```

```
fila = [4, 4]
```

Seguiremos o algoritmo descrito pela alternativa C, reproduzido a seguir, com suas respectivas modificações na situação das listas `pilha`, `pilhaAux` e `fila`.

1) `push(pilhaAux, pop(pilha))`

Este comando retira o elemento do último índice da pilha e o insere no topo da pilha auxiliar. Com isso, temos o que segue.

```
pilha = [5, 8, 3, 8]
```

```
pilhaAux = [5]
```

```
fila = [4, 4]
```

2) `enqueue(fila, pop(pilha))`

Este comando retira o elemento do último índice da pilha e o insere no fim da fila.

```
pilha = [5, 8, 3]
```

```
pilhaAux = [5]
```

```
fila = [8, 4, 4]
```

3) `enqueue(fila, pop(pilhaAux))`

Este comando retira o elemento do último índice da pilha auxiliar e o insere no fim da fila.

`pilha = [5, 8, 3]`

$$\text{pilAux} = []$$

```
fila = [5, 8, 4, 4]
```

4) pop(pilha)

Este comando retira o elemento de topo da pilha.

pilha = [5, 8]

`pilAux = []`

```
fila = [5, 8, 4, 4]
```

5) push(pilha,dequeue(fila))

Este comando remove o elemento de índice mais alto da fila e o coloca no topo da pilha.

`pilha = [5, 8, 4]`

`pilAux = []`

```
fila = [5, 8, 4]
```

Com isso, chegamos a duas listas idênticas, referenciadas pelas variáveis pilha e fila.

[illegible]