

Alexandre dos Santos Mignon

Pensamento computacional

Dados Internacionais de Catalogação na Publicação (CIP)
(Claudia Santos Costa - CRB 8^a/9050)

Mignon, Alexandre dos Santos

Pensamento computacional / Alexandre dos Santos Mignon. – São Paulo : Editora Senac São Paulo, 2025. – (Série Universitária)

Bibliografia

e-ISBN 978-85-396-5524-3 (ePub/2025)
e-ISBN 978-85-396-5525-0 (PDF/2025)

1. Programação (Computadores). 2. Desenvolvimento de software.
3. Algoritmos. 4. Python (Linguagem de programação). I. Título.
II. Série.

25-2453c

CDD – 005.1
BISAC COM051300
COM051360

Índice para catálogo sistemático:

- 1. Programação (Computadores) : Algoritmos 005.1**
2. Python (Linguagem de programação) 005.13

PENSAMENTO COMPUTACIONAL

Alexandre dos Santos Mignon





Administração Regional do Senac no Estado de São Paulo

Presidente do Conselho Regional

Abram Szajman

Diretor do Departamento Regional

Luiz Francisco de A. Salgado

Superintendente Universitário e de Desenvolvimento

Luiz Carlos Dourado

Editora Senac São Paulo

Conselho Editorial

Luiz Francisco de A. Salgado
Luiz Carlos Dourado
Darcio Sayad Maia
Lucila Mara Sbrana Sciotti
Luís Américo Tousi Botelho

Gerente/Publisher

Luís Américo Tousi Botelho

Coordenação Editorial

Verônica Marques Pirani

Prospecção

Andreza Fernandes dos Passos de Paula
Dolores Crisci Manzano
Paloma Marques Santos

Administrativo

Marina P. Alves

Comercial

Aldair Novais Pereira

Comunicação e Eventos

Tania Mayumi Doyama Natal

Coordenação de Arte

Antonio Carlos De Angelis

Coordenação de Revisão de Texto

Marcelo Nardeli

Preparação e Revisão de Texto

Cibele Machado

Acompanhamento Pedagógico

Otacilia da Paz Pereira

Designer Educacional

Priscila Arruda Ferreira de Carvalho

Revisão Técnica

Gustavo Moreira Calixto

Projeto Gráfico

Alexandre Lemes da Silva
Emilia Corrêa Abreu

Capa

Antonio Carlos De Angelis

Editoração Eletrônica e Ilustrações

Tiago Filu

Imagens

Adobe Stock Photos

Proibida a reprodução sem autorização expressa.
Todos os direitos desta edição reservados à

Editora Senac São Paulo
Av. Engenheiro Eusébio Stevaux, 823 – Prédio Editora
Jurubatuba – CEP 04696-000 – São Paulo – SP
Tel. (11) 2187 4450
editora@sp.senac.br
<https://www.editorasenacsp.com.br>

© Editora Senac São Paulo, 2025

Sumário

Capítulo 1 Introdução ao pensamento computacional, 7

- 1 Conceitos de pensamento computacional, 8
 - 2 Pilares do pensamento computacional, 10
 - 3 Algoritmos, 12
 - 4 Exemplos, 13
- Considerações finais, 16
Referências, 17

Capítulo 2 Introdução ao desenvolvimento de software, 19

- 1 Organização básica de um computador, 21
 - 2 Formas de representação de algoritmos, 23
 - 3 Processo simples de desenvolvimento de software, 27
- Considerações finais, 29
Referências, 29

Capítulo 3 Tipos de dados, variáveis e constantes, 31

- 1 Introdução à linguagem Python, 32
 - 2 Tipos de dados, 35
 - 3 Variáveis, 37
 - 4 Constantes, 40
 - 5 Operador de atribuição, 42
- Considerações finais, 43
Referências, 44

Capítulo 4 Estrutura sequencial, 45

- 1 Comandos de saída de dados, 46
 - 2 Comandos de entrada de dados, 51
- Considerações finais, 53
Referências, 53

Capítulo 5 Operadores aritméticos, 55

- 1 Operadores aritméticos, 56
 - 2 Expressões aritméticas, 57
 - 3 Funções matemáticas, 59
 - 4 Transformando expressões matemáticas, 62
 - 5 Operadores aritméticos de atribuição, 63
- Considerações finais, 64
Referências, 65

Capítulo 6 Operadores relacionais e lógicos, 67

- 1 Operadores relacionais, 68
 - 2 Operadores lógicos, 70
 - 3 Expressões lógicas, 73
- Considerações finais, 78
Referências, 78

Capítulo 7 Estruturas condicionais, 79

- 1 Estrutura condicional simples, 80
 - 2 Estrutura condicional composta, 83
 - 3 Indentação, 86
 - 4 Estrutura condicional aninhada e encadeada, 88
- Considerações finais, 92
Referências, 93

Capítulo 8 Estruturas de repetição, 95

- 1 Estrutura de repetição while, 96
 - 2 Estrutura de repetição for, 103
 - 3 Controle de fluxo com as instruções break e continue, 106
- Considerações finais, 108
Referências, 109

Sobre o autor, 111

Capítulo 1

Introdução ao pensamento computacional

Em um mundo cada vez mais dinâmico e orientado pela tecnologia, a capacidade de resolver problemas de forma eficaz tornou-se uma habilidade essencial. Quando nos deparamos com desafios complexos, contar com métodos estruturados para analisar, organizar e propor soluções inovadoras faz toda a diferença. Nesse contexto, o pensamento computacional surge como uma abordagem poderosa, pois permite decompor problemas, identificar padrões e desenvolver soluções sistemáticas. Embora frequentemente associado à programação, seu alcance vai muito além da ciência da computação, sendo aplicável a diversas áreas do conhecimento e contribuindo para a tomada de decisões estratégicas em diferentes profissões.

Para compreender melhor essa abordagem, podemos compará-la a um quebra-cabeça: inicialmente, a grande quantidade de peças espalhadas pode parecer confusa e desafiadora, mas, ao organizá-las e identificar padrões, a solução começa a emergir. O pensamento computacional segue essa mesma lógica, permitindo lidar com a complexidade ao dividir grandes desafios em partes menores e mais gerenciáveis. Esse raciocínio estruturado não apenas facilita a resolução de problemas na área tecnológica, mas também se revela valioso em setores como educação, saúde, negócios e engenharia, onde a análise crítica e a organização das informações são fundamentais para a identificação de soluções eficazes.

Com a crescente digitalização da sociedade, adotar essa forma de pensar torna-se um diferencial estratégico. Desenvolver a habilidade de enxergar problemas de maneira lógica e estruturada amplia a capacidade de enfrentar desafios com maior eficiência. Mais do que uma técnica, trata-se de um novo paradigma para perceber oportunidades e tomar decisões assertivas, fundamental para aqueles que desejam se destacar em um mundo cada vez mais orientado pela informação e pela inovação.

1 Conceitos de pensamento computacional

O pensamento computacional pode ser definido como um conjunto de habilidades cognitivas que permitem formular problemas e encontrar soluções de modo que um computador – ou um agente lógico – possa executá-las de forma eficiente. No entanto, essa competência vai além da simples programação ou do domínio de linguagens específicas. Trata-se de um modo de pensar característico da ciência da computação, no qual desafios são traduzidos em etapas lógicas, padrões são identificados e soluções eficientes são desenvolvidas para posterior implementação.

A pesquisadora Jeannette Wing (2006), uma das principais referências na área, define o pensamento computacional como um processo cognitivo que permite formular problemas e desenvolver soluções que

possam ser representadas de forma executável por um agente de processamento de informações. Essa perspectiva amplia a compreensão do conceito, evidenciando que ele não se restringe à programação, mas se configura como um método estratégico de raciocínio aplicável a diferentes contextos.

Além do ambiente computacional, esse tipo de raciocínio é útil em diversas áreas do conhecimento e no cotidiano. Médicos recorrem a ele para diagnosticar doenças, engenheiros para projetar estruturas, e empreendedores para desenvolver novos produtos. Até mesmo tarefas comuns, como organizar um orçamento familiar ou planejar uma mudança de casa, podem se beneficiar desse modelo de pensamento. Isso ocorre porque o pensamento computacional nos capacita a estruturar problemas e resolvê-los de maneira eficiente.

Esse processo envolve três etapas principais: (1) **analisar o problema**, identificando suas partes e definindo claramente os desafios envolvidos; (2) **formular uma solução**, elaborando um plano estruturado para alcançar o resultado desejado; e (3) **expressar essa solução de forma clara**, de modo que outra pessoa – ou um computador – possa comprehendê-la e executá-la.

Outro aspecto essencial do pensamento computacional é sua contribuição para o aprendizado contínuo. Ao estruturar e sequenciar ações de maneira lógica, ele permite maior clareza sobre os passos necessários para aprimorar qualquer atividade. Seja na aprendizagem de um novo idioma ou no desenvolvimento de software, a organização dos processos facilita a adaptação e o aperfeiçoamento ao longo do tempo.

Diante de sua relevância, o pensamento computacional tem sido cada vez mais integrado à educação, desde o ensino básico até o superior. Preparar estudantes para o mundo digital e para as exigências do mercado de trabalho significa capacitá-los a estruturar soluções de forma lógica, otimizando processos e reduzindo ambiguidades na tomada de

decisões. Assim, essa habilidade se consolida como essencial para enfrentar os desafios do século XXI.



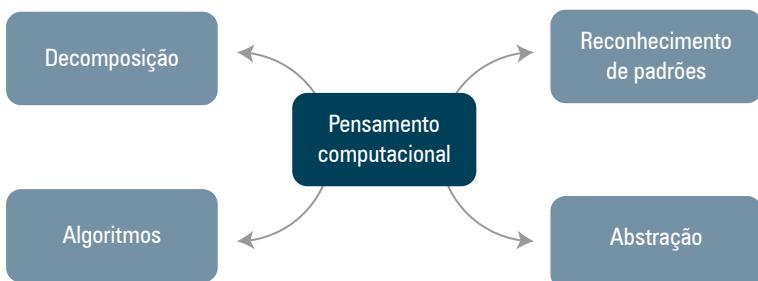
PARA SABER MAIS

Pesquise sobre “computação desplugada” na internet. Essa abordagem oferece maneiras práticas e acessíveis de desenvolver o pensamento computacional sem a necessidade de um computador, permitindo que qualquer pessoa compreenda os princípios fundamentais da computação de forma lúdica e interativa.

2 Pilares do pensamento computacional

Para compreender o pensamento computacional, é essencial conhecer seus quatro pilares fundamentais conforme proposto por Wing (2006). Eles estão ilustrados na figura 1 e descritos a seguir.

Figura 1 – Pilares do pensamento computacional



- **Decomposição:** consiste em dividir um problema complexo em partes menores e mais fáceis de resolver. Esse processo facilita a análise e a resolução, permitindo que cada componente seja tratado separadamente. Por exemplo, ao escrever um livro, a tarefa pode parecer desafiadora. No entanto, ao subdividi-la em capítulos, seções e parágrafos, torna-se mais viável. Esse princípio

também se aplica a problemas técnicos e científicos: ao fragmentá-los, é possível compreender melhor suas particularidades e desenvolver soluções mais eficazes.

- **Reconhecimento de padrões:** refere-se à identificação de semelhanças e regularidades em dados ou situações, possibilitando a formulação de soluções eficientes. No trânsito, por exemplo, a análise de padrões permite prever horários de pico e otimizar rotas. No contexto financeiro, a detecção de tendências é essencial para a tomada de decisões estratégicas. Esse pilar possibilita a reutilização de soluções já aplicadas em problemas semelhantes, reduzindo o tempo e o esforço necessários para resolver desafios complexos.
- **Abstração:** envolve a seleção dos elementos essenciais de um problema, ignorando informações irrelevantes. Um exemplo claro é um mapa: ele omite detalhes como a vegetação ou a cor das construções, focando apenas nos elementos essenciais para navegação, como ruas e pontos de referência. No pensamento computacional, esse processo permite simplificar problemas e concentrar-se nos aspectos fundamentais para sua resolução. Em uma viagem de carro, por exemplo, informações como distância, pedágios e postos de gasolina são relevantes, enquanto detalhes secundários, como a paisagem ao longo do percurso, podem ser desconsiderados.
- **Algoritmos:** refere-se à definição de uma sequência estruturada de passos para solucionar um problema. Um algoritmo deve apresentar instruções claras e organizadas para que sua execução ocorra de forma eficiente e sem ambiguidades. Embora sejam a base da programação de computadores, também estão presentes em diversas atividades do dia a dia, como planejar uma viagem ou montar um móvel seguindo um manual. Criar algoritmos eficientes permite otimizar processos e encontrar soluções adequadas.

Compreender e aplicar os pilares do pensamento computacional permite aprimorar a forma como problemas são analisados e resolvidos. Mais do que uma ferramenta da computação, essa abordagem se mostra essencial para o raciocínio lógico e a busca por soluções eficientes em diferentes áreas do conhecimento e do cotidiano.

3 Algoritmos

Os algoritmos são a materialização do pensamento computacional. Eles representam o passo a passo detalhado que leva à resolução de um problema específico, funcionando como uma receita de bolo: se cada instrução for seguida corretamente, o resultado será consistente, independentemente de quem esteja executando as etapas.

Um algoritmo pode ser definido como uma sequência finita de instruções bem definidas que levam à resolução de um problema (Forbellone; Eberspächer, 2022). Em outras palavras, é um conjunto de passos lógicos destinados a alcançar um objetivo. Para ilustrar esse conceito, apresentamos dois exemplos de algoritmos descritos em linguagem natural.

O primeiro exemplo mostra um algoritmo para calcular a média das notas de uma turma:

1. Somar todas as notas obtidas pelos alunos.
2. Contar o número total de alunos na turma, isto é, determinar o número total de notas.
3. Dividir a soma total das notas pelo número de alunos.
4. Apresentar o resultado dessa divisão, correspondente à média das notas.

Já o segundo exemplo apresenta um algoritmo para verificar se um número é par ou ímpar:

1. Obter o número a ser analisado.
2. Calcular o resto da divisão desse número por 2.
3. Se o resultado for zero, classificar o número como par.
4. Caso contrário, classificá-lo como ímpar.
5. Exibir a classificação correspondente (par ou ímpar).

Esses exemplos demonstram como a combinação de lógica, clareza e precisão pode ser aplicada para resolver problemas de forma estruturada, refletindo a essência do pensamento computacional.

A relação entre pensamento computacional e algoritmos é fundamental. Quando analisamos um problema utilizando os pilares da decomposição, reconhecimento de padrões e abstração, estruturamos a base para a criação de algoritmos eficientes. O pensamento computacional permite organizar e formular soluções de maneira lógica, enquanto os algoritmos formalizam essas soluções em sequências específicas de operações, tornando-as executáveis por um computador.



NA PRÁTICA

Imagine que Ana, uma estudante, precise organizar seu horário de estudos para cinco disciplinas diferentes. Um algoritmo para isso poderia incluir: listar as disciplinas, identificar a dificuldade de cada uma, alocar blocos de tempo compatíveis com a dificuldade, intercalar pausas para descanso e revisar o cronograma semanalmente.

4 Exemplos

Agora que exploramos os conceitos e pilares do pensamento computacional, vamos consolidar nosso aprendizado por meio de exemplos

práticos. Veremos como essa abordagem pode ser aplicada na resolução de problemas cotidianos e computacionais.

O nosso primeiro exemplo envolve a organização de uma viagem. Imagine que você está planejando uma viagem de férias. Esse é um problema que pode se beneficiar com a aplicação do pensamento computacional.

- Decomposição: primeiro, você divide o problema em partes menores: definir o destino, o período da viagem, o orçamento, o meio de transporte, a hospedagem, as atividades que deseja realizar, etc.
- Reconhecimento de padrões: você pode pesquisar por padrões em viagens anteriores ou em relatos de outras pessoas. Por exemplo, você percebe que viajar na baixa temporada é geralmente mais barato, ou que reservar hotéis com antecedência pode garantir melhores preços.
- Abstração: você se concentra nos aspectos mais importantes para o planejamento da viagem, como a documentação necessária, as reservas a serem feitas, e o roteiro principal, deixando detalhes menos relevantes para depois.
- Algoritmo: você cria um plano de ação, um passo a passo para organizar a viagem:
 1. Definir o destino e o período da viagem.
 2. Pesquisar e comparar preços de passagens e hospedagem.
 3. Definir um orçamento e estimar os gastos.
 4. Reservar as passagens e a hospedagem.
 5. Pesquisar sobre as atrações turísticas do destino.
 6. Montar um roteiro diário, definindo as atividades que deseja realizar em cada dia.

7. Fazer uma lista do que levar na mala.
8. Verificar a documentação necessária (passaporte, visto, etc.).

O nosso segundo exemplo tem como objetivo buscar uma receita na internet. Vamos considerar um exemplo mais simples: você decide fazer um bolo, mas não sabe a receita. Você recorre a um site de busca na internet. Como o pensamento computacional está presente nesse processo?

- **Decomposição:** o problema é dividido em etapas: definir o tipo de bolo que deseja fazer, digitar as palavras-chave na barra de busca, analisar os resultados, escolher uma receita e executá-la.
- **Reconhecimento de padrões:** você reconhece que os sites de busca utilizam algoritmos para encontrar páginas relevantes com base nas palavras-chave digitadas. Você também reconhece padrões nas receitas, como a presença de ingredientes comuns (farinha, ovos, açúcar) e uma estrutura similar (misturar ingredientes secos, adicionar líquidos, assar).
- **Abstração:** você foca nas informações essenciais da receita: ingredientes, quantidades e modo de preparo. Você ignora informações irrelevantes, como a história do bolo ou a biografia do autor da receita.
- **Algoritmo:** o site de busca utiliza um algoritmo complexo para classificar as páginas da internet e exibir os resultados mais relevantes para a sua busca. Esse algoritmo leva em conta diversos fatores, como a frequência das palavras-chave, a qualidade do conteúdo e a autoridade do site.

Para finalizar, vamos a um exemplo simples, relacionado com a área da computação. Imagine que você precisa encontrar o maior número em uma lista de números desordenados. Como você faria isso? Vamos aplicar o pensamento computacional para desenvolver um algoritmo para resolver esse problema.

- Decomposição: dividimos o problema em tarefas menores, comparar números e guardar o maior encontrado até o momento.
- Reconhecimento de padrões: percebemos que precisamos repetir a comparação para cada número da lista.
- Abstração: focamos no essencial: comparar dois números e lembrar do maior. Não precisamos nos preocupar com a posição dos números na lista, por exemplo.
- Algoritmo:
 1. Considere que o primeiro número da lista é o maior.
 2. Para cada número restante na lista, faça:
 - 2.1. Compare o número atual com o maior número encontrado até agora.
 - 2.2. Se o número atual for maior, atualize o maior número encontrado.
 3. Ao final, o maior número encontrado é o maior número da lista.

Esses exemplos ilustram como o pensamento computacional estrutura o raciocínio lógico, tornando problemas complexos mais fáceis de resolver. Seu uso vai muito além da computação, sendo aplicável a diversas áreas do conhecimento e do dia a dia.

Considerações finais

Ao longo deste capítulo, exploramos os conceitos e princípios do pensamento computacional, uma habilidade essencial no mundo digital. A capacidade de analisar problemas logicamente, dividi-los em partes menores, identificar padrões e desenvolver soluções eficientes é crucial em diversas áreas do conhecimento e do mercado de trabalho.

O principal objetivo desta obra é fortalecer o pensamento computacional e o raciocínio lógico por meio da criação de algoritmos e da programação de soluções simples. Ao aplicar os pilares desse conceito – decomposição, reconhecimento de padrões, abstração e construção de algoritmos – você desenvolverá estratégias mais eficazes para resolver problemas em diferentes contextos acadêmicos e profissionais.

Nos próximos capítulos, aprofundaremos a discussão sobre algoritmos e programação, proporcionando a oportunidade de aplicar os conceitos apresentados. O domínio do pensamento computacional não apenas aprimora a capacidade analítica e criativa, mas também fortalece competências essenciais para o futuro do trabalho, como a resolução estruturada de problemas e a inovação.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação: a construção de algoritmos e estruturas de dados com aplicações em Python.** 4. ed. São Paulo: Pearson; Porto Alegre: Bookman, 2022.

WING, Jeannette M. Computational thinking. **Communications of the ACM**, [s. l.], v. 49, n. 3, mar. 2006.

Capítulo 2

Introdução ao desenvolvimento de software

Os sistemas computacionais estão presentes em praticamente todos os aspectos do nosso cotidiano, desempenhando um papel essencial na realização de diversas tarefas, desde o envio de uma simples mensagem até cálculos científicos complexos. Para compreender seu funcionamento, é fundamental conhecer sua estrutura básica e como seus componentes interagem.

Todo sistema computacional é formado por hardware e software. O hardware refere-se à parte física do computador, ou seja, os componentes tangíveis, como processador, memória, disco rígido, teclado e monitor. Já o software representa a parte lógica do sistema, composta pelos programas e instruções que fazem o hardware funcionar, como sistemas operacionais e aplicativos. Ambos trabalham de forma integrada para garantir o processamento de informações e a execução de tarefas.

O funcionamento de um sistema computacional baseia-se na manipulação de dados e na execução de instruções. Os dados são informações brutas inseridas no sistema, como números, textos ou imagens, enquanto as *instruções* são comandos que o processador segue para processar esses dados e convertê-los em *informação*, ou seja, um conjunto de dados organizados e interpretados para gerar conhecimento ou apoiar decisões.

Por exemplo, ao usar uma calculadora digital, o usuário insere números (dados) e escolhe uma operação matemática (instrução). O processador executa essa operação e retorna o resultado (informação), que é exibido na tela.

Esse processamento segue um ciclo básico composto por três etapas principais:

1. **Entrada:** o sistema recebe informações externas, por exemplo, do usuário do sistema.
2. **Processamento:** o processador interpreta as instruções fornecidas pelo software e manipula os dados de acordo com regras predefinidas.
3. **Saída:** o resultado do processamento é apresentado ao usuário.

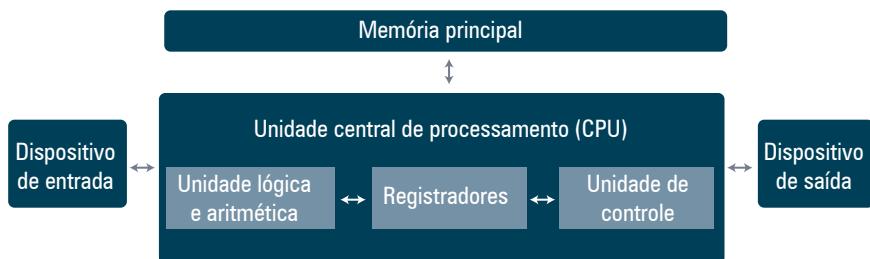
Esse ciclo se repete continuamente em qualquer sistema computacional, garantindo que as operações sejam realizadas de maneira eficiente e organizada. Compreender esse processo é essencial para o desenvolvimento de software.

1 Organização básica de um computador

Antes de iniciarmos o desenvolvimento de algoritmos e a construção de programas, é fundamental compreender o funcionamento do computador. Esse conhecimento permite que o programador escreva códigos mais eficientes, otimize o uso de recursos e compreenda melhor como o hardware influencia a execução dos programas. Além disso, entender a organização do computador facilita a assimilação das estruturas e comandos das linguagens de programação. Nesta seção, apresentamos os principais elementos da organização de um computador.

Os computadores modernos seguem um modelo baseado na arquitetura de Von Neumann (Tanenbaum; Austin, 2013), que estrutura o sistema computacional em três componentes principais: *CPU, memória principal* e *dispositivos de entrada e saída*. Esses elementos trabalham de forma integrada para garantir o processamento e a execução dos programas, como ilustrado na figura 1.

Figura 1 – Estrutura dos componentes básicos de um computador



Fonte: adaptado de Manzano e Oliveira (2019).

A CPU (central processing unit, em português unidade central de processamento) é o “cérebro” do computador, responsável pelo processamento e execução das instruções dos programas. Ela pode ser subdividida em três partes principais:

- **Unidade de controle:** interpreta instruções e gerencia o fluxo de dados dentro da CPU.
- **Unidade lógica e aritmética (ULA):** executa operações matemáticas e lógicas.
- **Registradores:** pequenos espaços de memória de alta velocidade usados para armazenar temporariamente dados e instruções.

A *memória principal*, também chamada de *memória RAM* (*random access memory* – memória de acesso aleatório), armazena temporariamente os dados e instruções que estão sendo processados. Trata-se de uma *memória volátil*, ou seja, seus dados são perdidos assim que o computador é desligado ou o programa em execução é encerrado. Enquanto o programa está em funcionamento, a CPU acessa constantemente essa memória para obter e armazenar informações. Sua velocidade e capacidade influenciam diretamente no desempenho do sistema, pois ela atua como um espaço de trabalho temporário para os programas.

Além da memória RAM, o computador também utiliza memórias auxiliares, como memória cache (para acesso ultrarrápido a dados frequentemente usados) e memória secundária (como HDs e SSDs) para armazenar dados de forma permanente.

Os dispositivos de entrada e saída (I/O – input/output) permitem a comunicação do computador com o usuário e outros sistemas. Os dispositivos de **entrada** enviam informações para o sistema, enquanto os de **saída** apresentam os resultados do processamento. Exemplos incluem:

- **Entrada:** teclado, mouse, scanner, sensores, câmeras, microfones.
- **Saída:** monitor, impressora, alto-falantes.

A interação entre CPU, memória e dispositivos de entrada e saída garante o funcionamento do computador e a execução dos programas. Para ilustrar esse processo, considere o seguinte exemplo: ao digitar um

número em uma calculadora digital no computador, o teclado (dispositivo de entrada) transmite o dado para a memória RAM, que é obtido posteriormente pela CPU. A CPU realiza os cálculos necessários e, em seguida, armazena temporariamente o resultado na memória antes de enviá-lo ao monitor (dispositivo de saída) para ser apresentado ao usuário. Esse ciclo de interação ocorre continuamente em todas as operações computacionais e sua eficiência depende da velocidade da CPU e da memória principal.

2 Formas de representação de algoritmos

A construção de algoritmos é um dos fundamentos essenciais da computação e da programação. Um algoritmo é uma sequência finita de instruções bem definidas que levam à solução de um problema (Forbel-lone; Eberspächer, 2022). Para representar essa sequência de forma comprehensível tanto para humanos quanto para máquinas, utilizam-se diferentes métodos. As principais formas de representação de algoritmos são descrição narrativa, fluxograma e pseudocódigo.

A seguir, apresentamos essas formas de representação por meio de um exemplo prático: *desenvolver um algoritmo que leia um número inteiro, calcule e exiba o dobro desse número.*

2.1 Descrição narrativa

A descrição narrativa é a forma mais simples de representar um algoritmo. Consiste na exposição dos passos necessários para a execução de uma tarefa utilizando linguagem natural. Essa abordagem é intuitiva e facilita o entendimento inicial de algoritmos, sendo especialmente útil para iniciantes. No entanto, sua principal limitação é a falta de formalidade e precisão, o que pode gerar ambiguidades em problemas mais complexos. A seguir apresentamos, em descrição narrativa, o algoritmo para calcular o dobro de um número.

1. Ler um número inteiro fornecido pelo usuário.
2. Calcular o dobro desse número.
3. Exibir o resultado do cálculo na tela.

2.2 Fluxograma

O **fluxograma**, também chamado de diagrama de blocos, é uma representação gráfica do algoritmo que utiliza símbolos padronizados para ilustrar o fluxo de execução das operações. Esse modelo facilita a visualização da lógica do programa, sendo amplamente utilizado na fase de planejamento e documentação de sistemas.

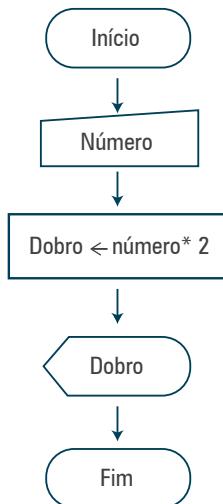
A figura 2 apresenta os principais símbolos utilizados em fluxogramas, e a figura 3 exibe a representação gráfica do algoritmo que calcula o dobro de um número.

Figura 2 – Símbolos do fluxograma

Símbolo	Descrição
	Terminal Representa o início e o fim do algoritmo.
	Processamento Representa a execução de operações ou ações.
	Teclado Representa entrada de dados por meio do teclado.
	Vídeo Representa a saída de informações em vídeo.
	Decisão Representa uma ação lógica que resulta na escolha de um conjunto de instruções.
	Preparação Representa uma ação de preparação para um processamento.
	Seta de orientação de fluxo Orienta a sequência de execução ou leitura.

Fonte: adaptado de Puga e Rissetti (2016).

Figura 3 – Fluxograma para calcular o dobro de um número



2.3 Pseudocódigo

O pseudocódigo é uma forma intermediária entre a linguagem natural e a programação. Ele utiliza uma sintaxe estruturada semelhante às linguagens de programação, mas sem seguir regras rígidas. O objetivo do pseudocódigo é tornar a lógica do algoritmo comprehensível para qualquer pessoa familiarizada com programação, independentemente da linguagem específica utilizada. A figura 4 apresenta o algoritmo, descrito em pseudocódigo, que calcula o dobro de um número.

Figura 4 – Pseudocódigo para calcular o dobro de um número

```
algoritmo CalculaDobro
var
    numero, dobro : inteiro
início
    escreva "Digite um número inteiro: "
    leia numero
    dobro ← numero * 2
    escreva "O dobro do número é: ", dobro
fim
```



PARA SABER MAIS

O Scratch é uma ferramenta de programação visual desenvolvida pelo MIT (Massachusetts Institute of Technology), voltada para crianças e iniciantes. Ele permite criar projetos interativos por meio de blocos coloridos que se encaixam como peças de um quebra-cabeça. Dessa forma, ensina conceitos fundamentais de lógica de programação sem a necessidade de escrever código. Para saber mais, basta pesquisar por Scratch na internet.

2.4 Linguagem de programação

Após a definição do algoritmo em uma das formas de representação mencionadas, o próximo passo é sua implementação em uma linguagem de programação. As *linguagens de programação* permitem que os algoritmos sejam traduzidos para um formato executável pelo computador. Cada linguagem possui sua própria sintaxe e características, mas todas compartilham conceitos fundamentais, como variáveis, estruturas condicionais e laços de repetição. O processo de implementação de algoritmos é conhecido como codificação (Forbellone; Eberspächer, 2022).

Nesta obra, utilizamos a linguagem de programação Python para implementar os algoritmos devido à sua sintaxe simples e intuitiva, que facilita o aprendizado. Na maioria dos casos, os conceitos e algoritmos são apresentados diretamente nessa linguagem, sem necessidade de uma representação intermediária. A figura 5 ilustra a implementação do algoritmo em Python que calcula o dobro de um número.

Figura 5 – Código em Python para calcular o dobro de um número

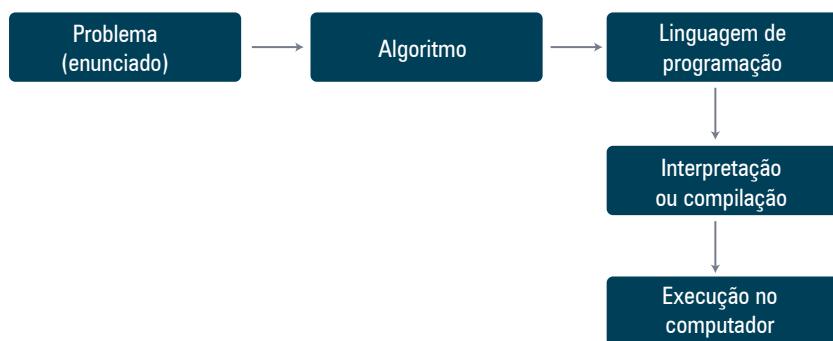
```
# Início do algoritmo
numero = int(input("Digite um número inteiro: ")) # Entrada de dados
dobro = numero * 2 # Processamento: cálculo do dobro
print("O dobro do número é: ", dobro) # Saída do resultado
```

As formas de representação de algoritmos desempenham um papel crucial na programação. A descrição narrativa auxilia no desenvolvimento do raciocínio inicial, os fluxogramas permitem visualizar a lógica do processo, e o pseudocódigo facilita a transição para uma linguagem de programação. Embora cada linguagem tenha sua própria sintaxe, os princípios fundamentais da lógica algorítmica são universais, permitindo a conversão eficiente entre essas formas de representação e o código-fonte final.

3 Processo simples de desenvolvimento de software

O desenvolvimento de um programa segue um processo estruturado composto por várias etapas. Nesta seção, descrevemos um processo simples que é utilizado ao longo desta obra. A figura 6 ilustra as etapas desse processo.

Figura 6 – Etapas de um processo simples de desenvolvimento de um programa



A primeira etapa no desenvolvimento de um programa é compreender claramente o problema a ser resolvido. Nessa fase, analisa-se o enunciado do problema para identificar os dados de entrada, o processamento necessário – o que o programa deve fazer, e os dados de saída (Ascencio; Campos, 2012).

Após essa análise, é criado um algoritmo que descreve de maneira estruturada os passos necessários para a solução do problema. Para facilitar a comunicação e a compreensão, esse algoritmo pode ser representado por diferentes formas, como pseudocódigo ou fluxograma.

Com o algoritmo definido, ele é convertido para uma linguagem de programação. Após a escrita do código, ele precisa ser traduzido para que o computador possa interpretá-lo ou executá-lo. Esse processo pode ocorrer de duas formas principais: compilação ou interpretação, dependendo da linguagem utilizada.

Em linguagens interpretadas, como Python e JavaScript, o código é executado linha por linha por um interpretador. Já em linguagens compiladas, como C, C# e Java, o código-fonte é convertido em um arquivo executável antes de ser executado no computador. Como nesta obra utilizamos a linguagem Python, o código será interpretado diretamente.

Por fim, com o código corretamente escrito e traduzido, o programa pode ser executado no computador.

Esse processo é fundamental para qualquer desenvolvimento de software, desde pequenos scripts até sistemas complexos. Seguir essas etapas garante que o código seja bem estruturado e funcional, contribuindo para a eficiência e manutenção do programa.



PARA SABER MAIS

Para desenvolver programas e resolver exercícios usando a linguagem de programação Python, recomenda-se pesquisar sobre como instalar e executar programas escritos em Python. A documentação oficial do Python e outros materiais acadêmicos podem auxiliar nesse processo.

Considerações finais

Neste capítulo foram apresentados os conceitos básicos do desenvolvimento de software, incluindo a estrutura de um computador e as formas de representação de algoritmos. A relação entre hardware e software foi discutida, destacando como esses elementos trabalham juntos para processar informações. Além disso, foram exploradas as principais formas de representar algoritmos, como descrição narrativa, fluxograma e pseudocódigo, mostrando suas aplicações na organização e planejamento de soluções computacionais.

O processo de desenvolvimento de software envolve etapas bem definidas, que começam com a análise do problema e seguem com a criação do algoritmo e sua implementação em uma linguagem de programação. A conversão do algoritmo em código-fonte permite que o computador interprete e execute as instruções corretamente. Seguir uma abordagem estruturada facilita a depuração, a manutenção e a evolução do software, tornando o código mais claro e eficiente.

Compreender esses fundamentos auxilia no aprendizado de programação e na construção de soluções mais organizadas. O conhecimento sobre algoritmos e sua implementação possibilita avanços no estudo de linguagens de programação e no desenvolvimento de sistemas. Os próximos capítulos aprofundarão esses temas, fornecendo ferramentas para a aplicação prática dos conceitos abordados.

Referências

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++ (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson, 2012.

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação:** a construção de algoritmos e estruturas de dados com aplicações em Python. 4. ed. São Paulo: Pearson; Porto Alegre: Bookman, 2022.

MANZANO, José Augusto N. G; OLIVEIRA, Jayr Figueiredo de. **Algoritmos**. 29. ed. São Paulo: Érica; Saraiva, 2019.

PUGA, Sandra Gavioli; RISSETTI, Gerson. **Lógica de programação e estruturas de dados:** com aplicações em Java. 3. ed. São Paulo: Pearson, 2016.

TANENBAUM, Andrew Stuart; AUSTIN, Todd. **Organização estruturada de computadores**. 6. ed. São Paulo: Pearson, 2013. *E-book*.

Capítulo 3

Tipos de dados, variáveis e constantes

A programação é uma habilidade essencial no mundo digital, possibilitando a criação de soluções automatizadas e inteligentes para diferentes contextos. Entre as diversas linguagens de programação disponíveis, Python se destaca por sua simplicidade e versatilidade, sendo amplamente utilizada tanto por iniciantes quanto por profissionais experientes. Este capítulo explora conceitos fundamentais dessa linguagem, com foco nos tipos de dados, variáveis e constantes, elementos essenciais para o desenvolvimento de algoritmos eficientes e bem estruturados.

Os tipos de dados representam as diferentes categorias de informações manipuladas dentro de um programa. Eles são fundamentais para garantir que as operações realizadas estejam de acordo com a natureza dos valores armazenados, evitando erros e otimizando o desempenho do código. Já as variáveis funcionam como espaços de armazenamento cujo conteúdo pode ser modificado ao longo da execução do programa, enquanto as constantes mantêm valores fixos, garantindo maior estabilidade e previsibilidade na implementação de algoritmos.

Além disso, o capítulo apresenta o operador de atribuição, um mecanismo essencial para definir e modificar valores armazenados em variáveis. Compreender esses conceitos é indispensável para que o estudante desenvolva programas sólidos e esteja preparado para desafios mais avançados em Python.

1 Introdução à linguagem Python

Python é uma das linguagens de programação mais populares e acessíveis da atualidade. Criada por Guido van Rossum no final dos anos 1980 e lançada oficialmente em 1991, Python foi projetada para ser simples, legível e de fácil aprendizado. Essa característica permite que até mesmo iniciantes consigam compreender conceitos de programação e algoritmos com mais facilidade (Python, 2025).

Trata-se de uma linguagem de alto nível, interpretada e com um forte foco na clareza do código. Python é amplamente utilizada em diversas áreas, como desenvolvimento web, análise de dados, automação de tarefas, inteligência artificial e computação científica. Sua filosofia de design enfatiza a legibilidade, a simplicidade e a facilidade de manutenção. A seguir, destacam-se algumas características que tornam essa linguagem especialmente atrativa para estudantes e profissionais:

- **Sintaxe simples e legível:** a estrutura sintática de Python é intuitiva, facilitando a escrita e manutenção de código.
- **Interpretada:** Python executa o código em tempo real, dispensando a necessidade de compilação prévia.
- **Tipagem dinâmica:** o tipo das variáveis é determinado automaticamente pelo interpretador.
- **Multiparadigma:** suporta programação orientada a objetos, funcional e imperativa, oferecendo flexibilidade ao desenvolvedor.

- **Multiplataforma:** compatível com Windows, MacOS, distribuições Linux e outros sistemas operacionais baseados no Unix, sem a necessidade de grandes adaptações.
- **Versatilidade:** utilizada em diversas áreas, como análise de dados, desenvolvimento web, aprendizado de máquina e automação.
- **Biblioteca padrão extensa:** possui inúmeros módulos nativos que facilitam o desenvolvimento de soluções diversas.
- **Comunidade ativa:** conta com um vasto suporte de desenvolvedores e materiais de estudo disponíveis on-line.

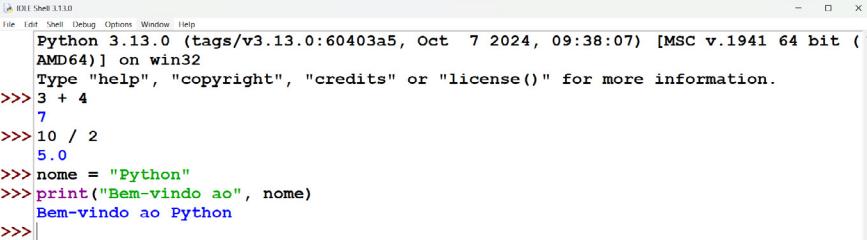
Para começar a programar em Python, é necessário instalar o interpretador da linguagem, que pode ser baixado no site oficial (Python, 2025). Além disso, existem diversos ambientes de desenvolvimento integrados (IDEs) que tornam a experiência mais eficiente, como o IDLE (fornecido junto com o Python), PyCharm, VS Code e Jupyter Notebook.

A execução de programas em Python pode ser feita *diretamente no terminal* – uma interface de linha de comando onde os usuários podem digitar comandos e executar programas, ou em um *ambiente interativo* – um local onde você pode executar comandos Python de forma interativa, ou seja, digitando código e vendo imediatamente a saída. No terminal, basta digitar `python` ou `python3` para iniciar o interpretador interativo onde comandos podem ser testados imediatamente. Para executar um script salvo em um arquivo `.py`, utilize o comando: `python nome_do_arquivo.py`.

O ambiente interativo do Python permite executar comandos em tempo real sem a necessidade de criar um arquivo de script. Esse recurso é útil para testar pequenas porções de código, explorar funcionalidades da linguagem e depurar erros rapidamente. Ele pode ser acessado de diferentes formas, sendo uma das principais o IDLE (integrated development and learning environment, em português: ambiente integrado de desenvolvimento e aprendizado), que é o ambiente oficial fornecido com

a instalação do Python. O IDLE é um editor leve e interativo que acompanha o Python por padrão. Ele permite escrever scripts e executar comandos diretamente no console interativo. A figura 1 apresenta um exemplo de uso do ambiente interativo do IDLE para executar comandos Python.

Figura 1 – Exemplo de uso do ambiente interativo do IDLE



```
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 3 + 4
7
>>> 10 / 2
5.0
>>> nome = "Python"
>>> print("Bem-vindo ao", nome)
Bem-vindo ao Python
>>>|
```

O IDLE também possui um editor de scripts (código-fonte de programas) no qual é possível escrever programas completos e executá-los. Isso permite criar programas mais elaborados sem a necessidade de digitar cada comando individualmente no console interativo. Com essa flexibilidade, o ambiente interativo do Python e o IDLE se tornam ferramentas essenciais para aprendizado, experimentação e desenvolvimento rápido de código.



NA PRÁTICA

Experimente criar e executar seu primeiro script Python! Para isso, certifique-se de que o interpretador Python está instalado em sua máquina. Em seguida, abra o IDLE. No IDLE, crie um arquivo acessando o menu: Arquivo > Novo Arquivo. No editor que se abrirá, digite o seguinte comando: `print("Olá, mundo!")`. Agora, salve o arquivo com o nome `meu_primeiro_programa.py` por meio do menu Arquivo > Salvar Como. Para executar o script, acesse Executar > Executar Módulo ou pressione F5. O resultado será exibido no console interativo do IDLE.

Além do IDLE, existem diversas IDEs (integrated development environment, em português: ambiente de desenvolvimento integrado) que oferecem recursos avançados para facilitar a programação em Python. Algumas das mais populares são o PyCharm, o Visual Studio Code (VS Code) e o Jupyter Notebook. Essas IDEs proporcionam funcionalidades como realce de sintaxe, autocompletar, depuração (*debugging*) e integração com controle de versão, tornando o desenvolvimento mais produtivo e eficiente.

2 Tipos de dados

Quando pensamos no funcionamento de um computador, podemos visualizá-lo como um ciclo contínuo de entrada, processamento e saída de dados. Nesse ciclo, o usuário fornece informações por meio de dispositivos de entrada (como teclado ou mouse), o computador processa esses dados seguindo instruções específicas e, por fim, exibe os resultados através de dispositivos de saída, como um monitor ou impressora.

Na programação, esse processo se reflete na manipulação de dados, que podem assumir diferentes formas, como números, textos e valores lógicos. Para que o computador possa gerenciar esses dados de maneira eficiente, foram estabelecidas categorias conhecidas como *tipos de dados*.

Os tipos de dados são um conceito essencial na programação. Eles definem quais valores podem ser armazenados e quais operações podem ser realizadas sobre eles. Por exemplo, se um dado for do tipo numérico inteiro, podemos utilizá-lo em operações aritméticas, como soma e subtração. No entanto, não podemos manipulá-lo como um texto.

O uso adequado dos tipos de dados melhora a eficiência, a legibilidade e a segurança do código. Ao definir corretamente os tipos, evitamos erros comuns, como tentar realizar operações incompatíveis entre

diferentes categorias de dados, o que pode resultar em falhas na execução do programa.

Cada linguagem de programação tem seu próprio conjunto de tipos de dados, mas algumas categorias fundamentais são comuns à maioria delas. A seguir, exploramos os tipos básicos usados na construção de algoritmos e como são implementados na linguagem Python.

Os tipos básicos, também chamados de *tipos primitivos*, são essenciais para a construção de qualquer programa. Eles incluem (Forbellone; Eberspächer, 2022):

- **Inteiro**: representa números inteiros: positivos, negativos ou zero. Exemplos: -10, 0, 1000.
- **Real**: representa números decimais (reais): positivos, negativos ou zero. Exemplos: -2.5, 0, 3.14.
- **Lógico**: representa valores lógicos (booleanos) que podem ser “verdadeiro” ou “falso”.
- **Texto**: representa sequências de caracteres. Esse tipo é chamado de string. Exemplo: “Olá, Mundo!”.

A tabela 1 apresenta a forma de representação dos tipos primitivos na linguagem de programação Python.

Tabela 1 – Representação dos tipos primitivos em Python

TIPO PRIMITIVO	REPRESENTAÇÃO EM PYTHON
Inteiro	int
Real	float
Lógico	bool
Texto	str (string)

Em Python, uma *string* é uma sequência imutável de caracteres usada para representar texto. Strings podem ser delimitadas por apóstrofo ('exemplo'), aspas ("exemplo") ou três aspas seguidas ("""exemplo"""), sendo esta última útil para textos de múltiplas linhas.

A imutabilidade das strings significa que, uma vez criada, seu conteúdo não pode ser alterado diretamente. Qualquer modificação gera uma nova string na memória, e não uma alteração na original. Esse comportamento evita mudanças inesperadas e melhora a segurança do código. A figura 2 apresenta exemplos de representação de strings em Python.

Figura 2 – Exemplos de representação de strings em Python

```
texto1 = "Olá, Mundo!"  
texto2 = 'Python é incrível!'  
texto3 = """Esta é  
uma string  
multilinha."""
```

3 Variáveis

A noção de variáveis é um dos pilares da programação. Uma variável pode ser entendida como um “armazenador” de valores que o programa manipula ao longo de sua execução. Em outras palavras, uma variável permite que o programador nomeie e guarde dados que poderão ser lidos ou modificados conforme a necessidade.

Uma variável, em termos gerais, é um espaço de memória identificado por um nome que o programador escolhe. Esse nome (também chamado de identificador) tem a função de facilitar o entendimento e a manipulação dos dados no código. A associação entre o nome e o valor é feita por meio de atribuição, permitindo que o valor possa ser acessado e modificado ao longo do programa.

Muitas linguagens de programação exigem que o programador declare antecipadamente o tipo de dado que será armazenado, como números inteiros, números de ponto flutuante ou textos. No entanto, Python adota um modelo de *tipagem dinâmica*, o que significa que não é necessário definir o tipo da variável antes de usá-la. O interpretador do Python determina automaticamente o tipo do dado com base no valor atribuído.

Para declarar uma variável em Python, definimos um identificador (nome da variável) e, em seguida, atribuímos um valor a ele. Em Python, a atribuição é feita de maneira simples, usando o operador `=`.

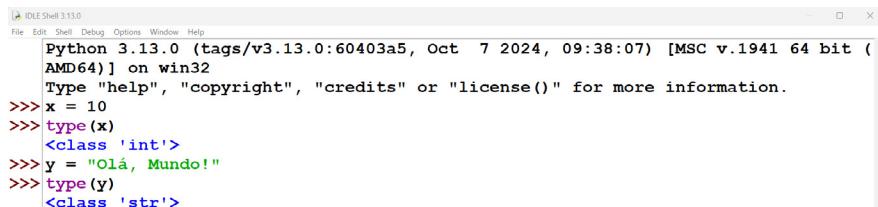
A figura 3 apresenta exemplos de definição de variáveis. Nesse caso, nome, idade e altura são variáveis que armazenam, respectivamente, uma string (texto), um número inteiro e um número de ponto flutuante. Observe que não precisamos especificar o tipo de cada variável. Essa flexibilidade do Python ajuda a tornar o desenvolvimento mais ágil, mas também exige atenção para lidar corretamente com diferentes tipos de dados.

Figura 3 – Exemplos de definição de variáveis

```
nome = "João"  
idade = 35  
altura = 1.78
```

Para saber qual é o tipo de dado que uma variável armazena, podemos usar a função interna `type()`. Esse comando é especialmente útil quando precisamos confirmar se a variável de fato possui o tipo esperado ou antes de aplicar certas operações que só funcionam com tipos específicos. A figura 4 apresenta um exemplo de uso do comando `type()`.

Figura 4 – Exemplo de uso do comando type()



A screenshot of the Python IDLE Shell window. The title bar says "IDLE Shell 3.13.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows Python version information and a few lines of code demonstrating the use of the type() function:

```
Python 3.13.0 (tags/v3.13.0:60403a5, Oct  7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> x = 10
>>> type(x)
<class 'int'>
>>> y = "Olá, Mundo!"
>>> type(y)
<class 'str'>
```

Ao criar nomes de variáveis, é importante seguir algumas regras básicas que garantem a compatibilidade com o interpretador Python e evitam erros no programa:

1. Comece com uma letra ou sublinhado (*underscore*, `_`): não é permitido iniciar o nome de uma variável com um dígito.
2. Use apenas caracteres alfanuméricos e sublinhado: não utilize espaços em branco, traços, pontos ou outros caracteres especiais no nome da variável.
3. Sensibilidade a maiúsculas: Python diferencia letras maiúsculas e minúsculas, ou seja, `Numero` e `numero` são variáveis diferentes.
4. Evite usar palavras reservadas (*keywords*): não utilize nomes que fazem parte da sintaxe do Python, como `if`, `for`, `while`, `True`, `False`, `class`, `def`, entre outros.
5. Escolha nomes significativos: embora não seja uma “regra sintática” obrigatória, usar nomes que descrevam o propósito da variável torna o código mais compreensível. Por exemplo, use `media_salarial` em vez de `ms`.



NA PRÁTICA

Para verificar todas as palavras reservadas do Python, experimente executar os seguintes comandos no modo interativo do Python:

```
import keyword  
print(keyword.kwlist)
```

Além das regras sintáticas, há convenções que visam tornar o código mais padronizado e legível. Algumas orientações são:

- Use letras minúsculas e separe palavras com sublinhado (_). Exemplo: `minha_variavel`, `dados_usuarios`.
- Prefira nomes descritivos para tornar o código mais intuitivo. Exemplo: `media_notas` é mais compreensível que `m`.
- Evite caracteres especiais, mantendo apenas letras, números e sublinhado.

Seguir essas diretrizes ajuda a evitar erros e facilita a manutenção do código, tornando-o mais compreensível para outros programadores.

4 Constantes

Uma constante é um valor que permanece inalterado desde o início até o fim de um programa (ou de um trecho de código) e, inclusive, ao longo de sucessivas execuções desse mesmo programa (Forbellone; Eberspächer, 2022). Como exemplo, podemos citar:

- Constantes numéricas: 5, 2527, -0.58.
- Constantes de texto (string): "Olá, Mundo!", "Python".
- Constantes lógicas (booleanas): `True`, `False`.

Em muitas linguagens de programação, como C, C++ e Java, é possível definir constantes nomeadas usando identificadores que recebem um valor fixo. Essas linguagens possuem mecanismos que impedem a alteração desses valores após a atribuição inicial. Por essa razão, as constantes também são conhecidas como variáveis somente de leitura (Puga; Rissetti, 2016).

O uso de constantes traz diversas vantagens, como:

- Maior legibilidade: em vez de utilizar valores numéricos soltos no código, podemos atribuir nomes significativos que tornam o programa mais compreensível.
- Facilidade de manutenção: se um valor precisar ser alterado, basta modificar a constante em um único local, evitando múltiplas edições.
- Segurança: em algumas linguagens, uma constante impede alterações acidentais, reduzindo a chance de erros inesperados.

Apesar de Python não ter um mecanismo intrínseco que faça cumprir a imutabilidade de variáveis, existe uma convenção de nomenclatura para representar uma constante. Convencionou-se que o nome de uma constante deve ser escrito em letras maiúsculas, com palavras separadas por sublinhado (_) quando necessário. A figura 5 apresenta exemplos de definição de constante em Python.

Figura 5 – Exemplo de definição de constantes em Python

```
PI = 3.14159
RAIO_MAXIMO = 50
```

Esses identificadores em letras maiúsculas indicam a outros desenvolvedores que esses valores não devem ser alterados. Porém, vale ressaltar que isso é apenas uma convenção e não há nada que impeça um programador de modificar o valor em tempo de execução, diferente de

linguagens em que o compilador ou o interpretador proíbem explicitamente a reatribuição de constantes.

5 Operador de atribuição

Os operadores de atribuição permitem que valores sejam armazenados em variáveis, estabelecendo uma forma de guardar dados ao longo da execução do programa. Em outras palavras, por meio de operadores de atribuição, podemos associar um valor a um identificador que será utilizado posteriormente.

Em linguagens de programação imperativas, como C, Java e Python, a atribuição segue uma estrutura comum: o identificador no lado esquerdo do operador recebe o valor resultante de uma expressão no lado direito.

Em Python, o operador de atribuição é o sinal de igual (`=`). A forma mais básica de atribuição em Python é: `variável = expressão`. Onde:

- `variável`: é o identificador que armazenará o valor.
- `expressão`: pode ser um número, um texto, um cálculo matemático, uma chamada de função ou qualquer outra operação que retorne um resultado.

A figura 6 apresenta exemplos de uso do operador de atribuição. Note que, neste caso, a variável `média` recebe o valor do resultado do cálculo da média aritmética das variáveis `nota1` e `nota2`.

Figura 6 – Exemplos de uso do operador de atribuição

```
mensagem = "Olá, Python!"  
PI = 3.14159  
nota1 = 6.5  
nota2 = 8.0  
media = (nota1 + nota2) / 2
```

Python também permite que se faça *atribuições múltiplas* em uma única linha. Outra variação comum é a “troca” de valores entre duas variáveis sem a necessidade de uma variável temporária. Além disso, é possível atribuir o mesmo valor a várias variáveis simultaneamente, isto é denominado de *atribuição encadeada*. A figura 7 apresenta exemplos dessas formas de atribuição.

Figura 7 – Exemplos de atribuição múltipla e atribuição encadeada

```
# atribuição múltipla
a, b, c = 1, 2, 3

# troca de valores entre duas variáveis
x, y = 10, 20
x, y = y, x

# atribuição encadeada
i = j = k = 0
```

Considerações finais

Neste capítulo, exploramos conceitos fundamentais da programação em Python, incluindo tipos de dados, variáveis, constantes e o operador de atribuição. Esses elementos são essenciais para estruturar programas de forma eficiente, garantindo que os dados sejam armazenados e manipulados corretamente.

A tipagem dinâmica do Python oferece flexibilidade ao desenvolvimento, mas exige atenção para evitar erros na atribuição de valores, especialmente em operações envolvendo diferentes tipos de dados. Além disso, a distinção entre variáveis e constantes reforça a importância da organização no código, tornando-o mais legível, compreensível e fácil de manter.

Compreender esses fundamentos permite ao programador construir algoritmos estruturados e funcionais, preparando-o para desafios mais avançados, como estruturas de controle e funções. O domínio dessas bases não apenas facilita a escrita de código robusto, mas também aprimora o pensamento lógico e a capacidade de resolução de problemas, competências essenciais para o desenvolvimento de soluções computacionais eficazes.

A programação é uma habilidade que se fortalece com a prática. Aplicar os conceitos estudados por meio de exercícios e projetos práticos é fundamental para consolidar o aprendizado e avançar para desafios mais complexos.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados com aplicações em Python. 4. ed. São Paulo: Pearson; Porto Alegre: Bookman, 2022.

PYTHON. Homepage. **Python**, 2025. Disponível em: <https://www.python.org/>. Acesso em: 21 mar. 2025.

PUGA, Sandra Gavioli; RISSETTI, Gerson. **Lógica de programação e estruturas de dados**: com aplicações em Java. 3. ed. São Paulo: Pearson, 2016.

Capítulo 4

Estrutura sequencial

A programação de computadores envolve a definição de sequências lógicas de instruções para resolver problemas específicos. Essas instruções seguem padrões chamados de estruturas de controle. Uma das estruturas mais básicas e fundamentais é a estrutura sequencial.

A estrutura sequencial é um padrão de controle de fluxo onde as instruções são executadas uma após a outra, na ordem em que aparecem no código, sem desvios ou repetições. Ela é usada em problemas simples que exigem apenas a execução direta de comandos.

Um aspecto essencial dentro da estrutura sequencial são os comandos de entrada e saída de dados. Eles permitem que o programa interaja com o usuário, capturando informações e exibindo resultados. Neste capítulo, apresentamos o conceito de comandos de entrada e saída e sua aplicação utilizando a linguagem de programação Python.

1 Comandos de saída de dados

A saída de dados em programação é o mecanismo pelo qual um programa exibe informações para o usuário (Forbellone; Eberspächer, 2022). Essa comunicação pode ocorrer por diferentes meios, como o terminal do sistema operacional, arquivos ou interfaces gráficas. Nesta obra, o foco é na saída via terminal (console), que é a forma mais direta e didática de compreender o conceito.

Em Python, o comando primário para exibição de informações no console é `print()`. Essa função é a forma mais direta de saída de dados via terminal. Sua sintaxe é apresentada na figura 1. Seus principais parâmetros são:

- valor1, valor2, ...: valores que serão exibidos. Podem ser de diferentes tipos, como strings, números e variáveis.
- sep: define o separador entre os valores (padrão é um espaço " ").
- end: define o caractere de finalização da linha (padrão é uma quebra de linha "\n").
- file: especifica onde a saída será enviada (por padrão, para o terminal).
- flush: se True, força a saída imediata (útil para logs em tempo real).

Os parâmetros que possuem valores padrão, como `sep=" "` e `end="\n"`, são opcionais. Isso significa que, caso não sejam especificados, a

função `print()` usará esses valores automaticamente. No entanto, se necessário, podemos alterá-los para personalizar a formatação da saída.

Figura 1 – Sintaxe do comando de saída `print()`

```
print(valor1, valor2, ..., valorN, sep=' ', end='\n', file=sys.stdout, flush=False)
```

O comando `print()` pode ser usado para exibir uma mensagem simples no console. Se fornecermos um único argumento do tipo string, ele será impresso exatamente como está. Além disso, o comando adiciona automaticamente uma quebra de linha ao final da saída, qualquer comando subsequente aparecerá na linha seguinte. A figura 2 apresenta um exemplo de impressão de um texto simples no console.

Figura 2 – Comando para impressão de uma mensagem simples no console

```
print("Olá, mundo!")
```

Além de exibir textos fixos, a função `print()` também permite exibir o valor armazenado em uma variável. Isso é útil quando queremos mostrar dados processados pelo programa, tornando a saída dinâmica. Basta passar a variável como argumento da função. A figura 3 apresenta exemplos de impressão do conteúdo de variável.

Figura 3 – Exemplos de impressão de conteúdo de variável no console

```
nome = "Ana"  
print(nome)
```

```
idade = 25  
print(idade)
```

Podemos imprimir múltiplos valores na mesma linha, separando-os por vírgulas dentro da função `print()`. Por padrão, o Python insere um

espaço entre cada valor impresso. Isso é útil quando queremos exibir variáveis junto com textos explicativos sem precisar concatenar strings manualmente. A figura 4 apresenta um exemplo de exibição de múltiplos valores na mesma linha.

Figura 4 – Exemplo de impressão de múltiplos valores na mesma linha

```
nome = "Ana"  
idade = 25  
print("Nome:", nome, "Idade:", idade)
```

O parâmetro `sep` permite definir um separador personalizado entre os valores passados para a função `print()`. O valor padrão é um espaço (" "), mas podemos alterá-lo para qualquer caractere ou string, como um hífen, vírgula ou qualquer outro símbolo, dependendo do formato desejado. A figura 5 apresenta um exemplo onde o parâmetro `sep` foi alterado para um hífen.

Figura 5 – Exemplo de modificação do parâmetro `sep`

```
print("Python", "é", "incrível!", sep="-")
```

Por padrão, a função `print()` adiciona uma quebra de linha (\n) ao final da saída. No entanto, podemos modificar esse comportamento usando o parâmetro `end`. Na figura 6, definimos `end=" "`, o que substitui a quebra de linha por um espaço, permitindo que a próxima chamada de `print()` continue na mesma linha.

Figura 6 – Exemplo de modificação do parâmetro `end`

```
print("Linha 1", end=" ")  
print("Linha 2")
```

O método `.format()` permite inserir valores dentro de uma string de forma organizada e flexível. Ele substitui `{}` pelos valores passados como argumento, garantindo uma melhor legibilidade da saída. A figura 7 apresenta um exemplo de formatação de strings com o `format()`.

Figura 7 – Exemplo de formatação de strings com o método `.format()`

```
nome = "Ana"  
idade = 25  
print("Meu nome é {} e eu tenho {} anos.".format(nome, idade))
```

A partir do Python 3.6, podemos usar *f-strings*, uma maneira mais simples e direta de formatar strings. Basta adicionar um `f` antes da string e colocar as variáveis dentro de `{}`. As f-strings também permitem expressões dentro das chaves, como cálculos e chamadas de funções. A figura 8 apresenta exemplos do uso de *f-strings*.

Figura 8 – Exemplos de uso de f-strings

```
nome = "Ana"  
idade = 25  
print(f"Meu nome é {nome} e eu tenho {idade} anos."  
  
numero = 7  
print(f"O dobro de {numero} é {numero * 2}.")
```

A saída de dados pode ser formatada de diversas formas, especialmente quando trabalhamos com números decimais (`float`). O Python oferece diferentes maneiras de controlar o número de casas decimais exibidas, garantindo que os resultados apareçam de forma clara e padronizada.

Podemos imprimir diretamente um valor `float` sem qualquer formatação especial. Por padrão, o Python exibe todos os dígitos disponíveis no valor armazenado, o que pode variar dependendo da precisão usada nos cálculos.

A função `format()` permite especificar o número de casas decimais desejadas. Podemos usar `{:.2f}` dentro da string para garantir que um número `float` seja exibido com exatamente duas casas decimais. O código `{:.2f}` indica que queremos um número decimal com duas casas decimais, arredondando automaticamente o valor. A figura 9 apresenta um exemplo de formatação de casas decimais usando a função `format()`.

Figura 9 – Exemplo de formatação de duas casas decimais usando a função `format()`

```
preco = 49.98765
print("O preço do produto é: {:.2f}".format(preco))
```

As *f-strings* também permitem formatar números diretamente dentro da string. Basta utilizar `{variável:.2f}` dentro da string para garantir que o número seja exibido com a precisão desejada. Você pode alterar o número 2 para qualquer valor de casas decimais que desejar. A figura 10 apresenta um exemplo de formatação de casas decimais usando *f-strings*.

Figura 10 – Exemplo de formatação de casas decimais usando *f-strings*

```
preco = 49.98765
print(f'O preço do produto é: {preco:.2f}')
```



PARA SABER MAIS

Na linguagem Python há diversas maneiras de formatar textos e números, desde o uso do método `.format()` até as modernas *f-strings*, que tornam o código mais conciso e intuitivo. Para se aprofundar mais nesse tema, busque na internet por “formatação de strings em Python”.

2 Comandos de entrada de dados

Os comandos de entrada são fundamentais na comunicação entre o usuário e um programa de computador. Eles são instruções que solicitam dados ao usuário, permitindo que os programas recebam informações externas durante a execução (Forbellone; Eberspächer, 2022). Em Python, a principal forma de capturar dados via teclado é através da função `input()`.

A função `input()` pode receber uma string opcional como mensagem de prompt e retorna sempre os dados digitados pelo usuário como uma string. Para que o programa possa processar essas informações, o valor retornado deve ser armazenado em uma variável. A figura 11 ilustra a sintaxe básica da função `input()`.

Figura 11 – Sintaxe básica da função `input()`

```
variavel = input(mensagem)
```

A figura 12 apresenta um exemplo que solicita um nome ao usuário e exibe uma mensagem personalizada. Neste exemplo, a entrada do usuário é armazenada na variável `nome` e posteriormente utilizada na exibição da mensagem.

Figura 12 – Exemplo de uso da função `input()`

```
nome = input("Digite seu nome: ")
print(f"Olá, {nome}!")
```

Por padrão, a função `input()` retorna uma string, então é necessário converter para um tipo numérico ao lidar com números. A função `int()` converte uma string que representa um número inteiro em um valor do tipo inteiro em Python. A figura 13 apresenta um exemplo de entrada de dados do tipo inteiro.

Figura 13 – Exemplo de entrada de dados do tipo inteiro

```
idade = int(input("Digite a sua idade: "))
print(f"Você tem {idade} anos.")
```

A função `float()` converte a string que representa um número decimal (ponto flutuante) em um tipo numérico apropriado para lidar com casas decimais. A figura 14 apresenta um exemplo de entrada de dados do tipo real.

Figura 14 – Exemplo de entrada de dados do tipo real

```
salario = float(input("Digite o seu salário: "))
print(f"Seu salário é R$ {salario:.2f}.")
```



PARA SABER MAIS

Ao usar as funções `int()` ou `float()` para converter uma string em um valor numérico, é importante lembrar que, se o usuário digitar um valor não numérico, o programa gerará uma exceção. Para aprender como lidar com essa situação, pesquise na internet por “tratamento de exceções em Python”.

A figura 15 apresenta um exemplo de leitura de três tipos de dados: string, int e float. O programa solicita o nome, a idade e a altura do usuário, exibindo os valores de maneira formatada, clara e objetiva. Observe que a altura é exibida com uma casa decimal.

Figura 15 – Exemplo de entrada de dados com string, int e float

```
nome = input("Digite seu nome: ")
idade = int(input("Digite sua idade: "))
altura = float(input("Digite sua altura em metros: "))

print(f"Olá, {nome}! Você tem {idade} anos e {altura:.1f} metros de altura.")
```

Considerações finais

A estrutura sequencial é a mais simples entre as estruturas de controle, garantindo a execução linear de comandos. Em Python, sua implementação é direta, facilitando a escrita de códigos claros e funcionais. Apesar das limitações, é um conceito essencial que serve de base para o aprendizado de estruturas mais avançadas, como condições e laços de repetição. Além disso, compreender bem essa estrutura contribui para a construção de algoritmos mais eficientes e fortalece a lógica de programação.

O domínio dos comandos de entrada e saída de dados também é fundamental para desenvolver programas interativos. A capacidade de coletar informações do usuário e exibir resultados de forma estruturada melhora a usabilidade do software e amplia suas aplicações em áreas como automação, análise de dados e interfaces simples.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados com aplicações em Python. 4. ed. São Paulo: Pearson; Porto Alegre: Bookman, 2022.

Capítulo 5

Operadores aritméticos

Os operadores aritméticos são componentes fundamentais na programação, responsáveis por viabilizar a manipulação de valores numéricos e a construção de expressões matemáticas. Em Python, esses operadores permitem a realização de operações básicas, como soma e subtração, bem como cálculos mais sofisticados, incluindo exponenciação e divisões inteiras. Além disso, a linguagem oferece uma biblioteca matemática (math) que expande significativamente as possibilidades computacionais, fornecendo funções para cálculos estatísticos, trigonométricos e logarítmicos.

Este capítulo explora os operadores aritméticos disponíveis em Python, abordando como utilizá-los para criar expressões matemáticas e como converter expressões matemáticas convencionais em código Python. Também discutiremos o uso da biblioteca `math` para realizar operações mais avançadas.

1 Operadores aritméticos

Os operadores aritméticos são um conjunto de símbolos que representam as operações básicas da matemática (Forbellone; Eberspächer, 2022). A tabela 1 apresenta a relação dos operadores aritméticos da linguagem Python. Esses operadores são amplamente utilizados em diversos contextos computacionais, como cálculos financeiros, estatísticos, simulações científicas e análise de dados.

Tabela 1 – Operadores aritméticos da linguagem Python

OPERADOR	DESCRIÇÃO	EXEMPLO	RESULTADO
<code>+</code>	Adição	<code>3 + 2</code>	5
<code>-</code>	Subtração	<code>5 - 3</code>	2
<code>*</code>	Multiplicação	<code>4 * 2</code>	8
<code>/</code>	Divisão	<code>10 / 2</code>	5.0
<code>//</code>	Divisão inteira	<code>10 // 3</code>	3
<code>%</code>	Módulo (resto da divisão)	<code>10 % 3</code>	1
<code>**</code>	Exponenciação	<code>2 ** 3</code>	8

Observe que temos três operadores relacionados à divisão. O operador `/` realiza uma divisão tradicional entre dois números em Python e sempre retorna um valor do tipo `float` (número com ponto flutuante), mesmo quando a divisão for exata. O operador `//` realiza a divisão entre

dois números, mas retorna apenas a parte inteira do resultado, descartando a parte decimal. Ele é especialmente útil quando você precisa garantir que o resultado seja um número inteiro. E, o operador `%` retorna o resto da divisão entre dois números. É muito utilizado para identificar números pares ou ímpares, verificar múltiplos ou realizar operações cíclicas.

A seguir, apresentamos um exemplo de uso de operadores aritméticos baseado no seguinte enunciado: escreva um programa em Python que receba do usuário um valor inteiro representando uma quantidade de minutos. O programa deve converter esse valor em horas e minutos, exibindo-os separadamente. Por exemplo, caso o usuário informe 135 minutos, seu programa deverá exibir como resultado 2 horas e 15 minutos.

A figura 1 apresenta o código da solução. Para resolver esse problema, utilizamos os operadores de divisão inteira (`//`) e módulo (`%`). Primeiramente, realizamos a divisão inteira do total de minutos por 60, que nos dará a quantidade de horas completas. Em seguida, usamos o operador módulo (`%`) para obter o restante de minutos que não formam uma hora completa.

Figura 1 – Código para conversão de tempo de minutos para horas e minutos

```
minutos = int(input("Informe o número de minutos: "))

horas = minutos // 60
minutos_restantes = minutos % 60

print(f"{horas} horas e {minutos_restantes} minutos")
```

2 Expressões aritméticas

Uma expressão aritmética é uma combinação de operandos (valores ou variáveis) e operadores que, quando avaliada, produz um resultado

numérico (Xavier, 2018). Em Python, as expressões aritméticas seguem regras de precedência semelhantes às da matemática convencional:

1. **Parênteses ()** - têm a mais alta prioridade e definem explicitamente a ordem de execução das operações.
2. **Exponenciação **** - é avaliada antes de qualquer outra operação.
3. **Multiplicação *, divisão /, divisão inteira // e módulo %** - possuem prioridade intermediária.
4. **Adição + e subtração** - são avaliadas por último.

Operadores com a mesma ordem de precedência são avaliados da esquerda para a direita. Entretanto, o operador de exponenciação é avaliado da direita para a esquerda. A figura 2 apresenta um exemplo de expressão aritmética com vários operadores.

Figura 2 – Exemplo de expressão aritmética com vários operadores

```
resultado = 10 + 2 * 3 ** 2 / (4 - 2)
```

A avaliação desta expressão ocorre da seguinte forma:

1. Parênteses: $(4 - 2)$ resulta em 2;
2. Exponenciação: $3^{**} 2$ resulta em 9;
3. Multiplicação: $2 * 9$ resulta em 18;
4. Divisão: $18 / 2$ resulta em 9.0;
5. Adição: $10 + 9.0$ resulta em 19.0.

Assim, resultado será 19.0.

3 Funções matemáticas

A linguagem Python é amplamente utilizada em diversos campos da ciência e da engenharia devido à sua simplicidade e versatilidade. Uma das razões para essa popularidade é a robusta biblioteca padrão que Python oferece, incluindo o módulo `math`, que fornece um conjunto abrangente de funções matemáticas.

O módulo `math` é parte da biblioteca padrão do Python e oferece acesso a muitas funções matemáticas de ponto flutuante, como funções trigonométricas, exponenciais e logarítmicas. Para utilizar essas funções, é necessário importar o módulo utilizando o comando `import math`. A tabela 2 apresenta uma relação das principais funções contidas no módulo `math`.

Tabela 2 – Principais funções do módulo `math`

SINTAXE DA FUNÇÃO	DESCRIÇÃO
<code>math.sqrt(x)</code>	Retorna a raiz quadrada de x.
<code>math.pow(x, y)</code>	Calcula x elevado à potência y.
<code>math.exp(x)</code>	Retorna o número de Euler (e) elevado à potência x.
<code>math.log(x)</code>	Retorna o logaritmo natural (base e) de x.
<code>math.log(x, base)</code>	Retorna o logaritmo de x na base especificada (base).
<code>math.sin(x)</code>	Calcula o seno do ângulo x (em radianos).
<code>math.cos(x)</code>	Calcula o cosseno do ângulo x (em radianos).
<code>math.tan(x)</code>	Calcula a tangente do ângulo x (em radianos).
<code>math.degrees(x)</code>	Converte ângulo de radianos para graus.
<code>math.radians(x)</code>	Converte ângulo de graus para radianos.
<code>math.ceil(x)</code>	Retorna o menor inteiro maior ou igual a x (arredonda para cima).

(cont.)

SINTAXE DA FUNÇÃO	DESCRIÇÃO
math.floor(x)	Retorna o maior inteiro menor ou igual a x (arredonda para baixo).
math.fabs(x)	Retorna o valor absoluto (positivo) de x como ponto flutuante.
math.factorial(x)	Retorna o fatorial do número inteiro não negativo x.
math.trunc(x)	Remove a parte decimal de x, retornando apenas a parte inteira.

Vamos para um exemplo de uso de funções matemáticas. Imagine que precisamos calcular a distância horizontal alcançada por uma bola lançada a partir do solo, considerando que o lançamento foi realizado com uma velocidade inicial conhecida e um determinado ângulo em relação à horizontal. A fórmula da física para o cálculo do alcance horizontal de um projétil lançado com velocidade inicial e ângulo é dada por:

Dados do problema:

- Velocidade inicial do lançamento (): 25 m/s
- Ângulo de lançamento em relação ao solo (): 30 graus
- Aceleração da gravidade (): 9,8 m/s²

A figura 3 apresenta o código em Python para a resolução do problema. Como o ângulo é dado em graus foi necessário convertê-lo para radianos porque as funções trigonométricas do módulo `math` usam radianos como unidade padrão.

Figura 3 – Código para calcular o alcance horizontal de um projétil

```
import math

# Dados iniciais
velocidade_inicial = 25 # em m/s
angulo_graus = 30         # em graus
g = 9.8                   # gravidade em m/s²

# Convertendo o ângulo de graus para radianos
angulo_radianos = math.radians(angulo_graus)

# Calculando o alcance
alcance = (math.pow(velocidade_inicial, 2) * math.sin(2 * angulo_radianos)) / g

print(f'O alcance horizontal do projétil é {alcance:.2f} metros.')
```

Além das funções matemáticas, o módulo `math` também fornece constantes matemáticas amplamente utilizadas em cálculos científicos e engenharia. As constantes mais importantes são:

- A constante `math.pi` representa o valor de π (π), a razão entre a circunferência de um círculo e o seu diâmetro, com um valor aproximado de 3,141592653589793.
- A constante `math.e` corresponde ao número de Euler, que é a base dos logaritmos naturais. Seu valor aproximado é 2,718281828459045.

A figura 4 apresenta um programa para o cálculo da área e da circunferência de um círculo.

Figura 4 – Código em Python para calcular a área e a circunferência de um círculo

```
import math

raio = float(input("Raio: "))
circunferencia = 2 * math.pi * raio
area = math.pi * raio ** 2

print(f"Circunferência do círculo: {circunferencia}")
print(f"Área do círculo: {area}")
```



PARA SABER MAIS

Para conhecer todas as funções matemáticas e constantes pertencentes ao módulo math, busque por “Funções Matemáticas em Python” e consulte a documentação oficial.

4 Transformando expressões matemáticas

Na programação, muitas expressões matemáticas complexas precisam ser transformadas em expressões lineares, ou seja, reescritas de forma que a linguagem de programação consiga interpretá-las corretamente. Essa transformação deve levar em conta as regras de precedência dos operadores, a sintaxe da linguagem e a clareza do código.

Python segue a notação matemática tradicional, mas algumas operações precisam ser ajustadas para garantir que os cálculos sejam feitos corretamente. Ao converter expressões matemáticas para código, é importante lembrar:

1. Substituir operadores matemáticos por operadores Python (por exemplo, \div vira $/$, \times vira $*$).
2. Usar parênteses para garantir a ordem correta das operações.
3. Converter expressões com raízes, logaritmos e funções trigonométricas para suas equivalentes do módulo math.

A figura 5 apresenta o código em Python da expressão matemática apresentada abaixo. Observe que o uso de parênteses garante que a soma seja feita antes da divisão.

Figura 5 – Código em Python de uma expressão matemática

```
x = 5
y = (2 * x + 3) / 4
print(y) # Resultado: 2.75
```

5 Operadores aritméticos de atribuição

No desenvolvimento de software, é comum a necessidade de atualizar o valor de variáveis com base em operações matemáticas. Para simplificar e tornar o código mais legível, Python oferece os operadores aritméticos de atribuição. Os operadores aritméticos de atribuição combinam uma operação aritmética com a atribuição de valor a uma variável. Eles são utilizados para simplificar expressões que modificam o valor de uma variável com base nela mesma. Esses operadores tornam o código mais conciso e legível, reduzindo a repetição de expressões. A tabela 3 apresenta os operadores aritméticos de atribuição da linguagem Python.

Tabela 3 – Operadores aritméticos de atribuição

OPERADOR	EQUIVALENTE A	EXEMPLO (X = 10)	RESULTADO
<code>+=</code>	<code>x = x + y</code>	<code>x += 5</code>	<code>x = 15</code>
<code>-=</code>	<code>x = x - y</code>	<code>x -= 3</code>	<code>x = 7</code>
<code>*=</code>	<code>x = x * y</code>	<code>x *= 2</code>	<code>x = 20</code>
<code>/=</code>	<code>x = x / y</code>	<code>x /= 2</code>	<code>x = 5.0</code>
<code>//=</code>	<code>x = x // y</code>	<code>x //= 3</code>	<code>x = 3</code>
<code>%=</code>	<code>x = x % y</code>	<code>x %= 4</code>	<code>x = 1</code>
<code>**=</code>	<code>x = x ** y</code>	<code>x **= 2</code>	<code>x = 100</code>

Os operadores aritméticos de atribuição podem ser combinados em expressões mais complexas, permitindo realizar múltiplas operações de forma clara e concisa em uma única linha de código. A combinação desses operadores simplifica cálculos matemáticos complexos e ajuda a melhorar a legibilidade do código, especialmente em situações em que várias variáveis são atualizadas em sequência.

Por exemplo, considere que você tenha três variáveis `x`, `y` e `z`. Inicialmente, digamos que `x = 10`, `y = 5` e `z = 3`. Você pode realizar uma combinação de operações da seguinte maneira: `x += y - z`. Nesta expressão, primeiro é avaliado o lado direito: calcula-se `y - z` (que é $5 - 3 = 2$), e então o valor resultante é somado ao valor de `x`. Após essa operação, o valor de `x` será atualizado para 12 (pois $10 + 2 = 12$).

A figura 6 apresenta um exemplo de combinação de diversos operadores aritméticos de atribuição em sequência. Neste código, primeiro `x` é multiplicado pelo resultado da expressão (`y + z`) e atualizado para 48. Depois, `y` é atualizado com a divisão inteira por `z`, passando a valer 2. Por fim, `z` recebe o resto da divisão dele mesmo por `y`, resultando em 0.

Figura 6 – Exemplo de uso dos operadores aritméticos de atribuição

```
x = 8
y = 4
z = 2

x *= y + z # Primeiro calcula y + z (6), então multiplica por x (8 * 6 = 48)
y //= z     # Realiza divisão inteira de y por z (4 // 2 = 2)
z %= y      # Atualiza z com resto da divisão de z por y (2 % 2 = 0)

print(f"x = {x}, y = {y}, z = {z}")
```

Considerações finais

Neste capítulo, exploramos a importância dos operadores aritméticos na programação, com foco na linguagem Python. Abordamos a funcionalidade de cada operador, incluindo soma, subtração, multiplicação,

divisão e exponenciação, além de suas aplicações práticas em expressões matemáticas. Também analisamos a biblioteca math, que expande as possibilidades computacionais ao oferecer funções avançadas para cálculos científicos e estatísticos. Esses conceitos são fundamentais para a manipulação de dados numéricos e a construção de algoritmos eficientes.

Além disso, discutimos a relevância dos operadores aritméticos de atribuição, que simplificam a escrita e a compreensão do código ao combinar operações matemáticas com a atribuição de valores a variáveis. Essa abordagem contribui para a clareza e a concisão dos algoritmos, reduzindo redundâncias e facilitando a manutenção do código. Também exploramos a conversão de expressões matemáticas para Python, garantindo que sejam interpretadas corretamente pela linguagem e otimizando a lógica computacional.

Por fim, compreendemos que o domínio dos operadores aritméticos é essencial para o desenvolvimento de soluções eficientes em programação. Seja para cálculos simples ou para aplicações complexas, a correta utilização desses operadores permite criar algoritmos mais robustos e precisos. Com o conhecimento adquirido, o programador está mais preparado para aplicar operações matemáticas em diversos contextos computacionais.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados com aplicações em Python. 4 ed. São Paulo: Bookman, 2022.

XAVIER, Gley Fabiano Cardoso. **Lógica de programação**. 13. ed. São Paulo: Editora Senac São Paulo, 2018.

Capítulo 6

Operadores relacionais e lógicos

Os operadores relacionais e lógicos são componentes essenciais em qualquer linguagem de programação. Eles constituem a base para a maioria das decisões que um programa precisa tomar, permitindo comparar valores, testar condições e definir o rumo da execução com base nos resultados dessas comparações.

Neste capítulo, vamos explorar como esses operadores funcionam na linguagem Python, abordando desde os conceitos mais básicos — ideais para quem está começando — até aspectos mais avançados, que ajudam a lidar com expressões complexas. Serão apresentados exemplos práticos, boas práticas de escrita e situações reais em que esses operadores são utilizados.

Entender profundamente o uso de operadores relacionais e lógicos é crucial para escrever código eficiente, legível e confiável. Eles estão diretamente ligados ao funcionamento de estruturas de controle como `if` e `while`, e aparecem em quase todas as situações que envolvem lógica condicional. Ao dominar esses conceitos, você estará mais preparado para construir programas mais inteligentes, seguros e flexíveis.

1 Operadores relacionais

Operadores relacionais, também conhecidos como operadores de comparação, são utilizados para comparar dois valores ou expressões do mesmo tipo de dado (Forbellone; Eberspächer, 2022). O resultado dessa comparação é sempre um valor lógico (booleano): `True` (verdadeiro) ou `False` (falso). Em Python, os principais operadores relacionais são:

- **Igual a (==)**: verifica se dois valores são iguais;
- **Diferente de (!=)**: verifica se dois valores são diferentes;
- **Maior que (>)**: verifica se o valor à esquerda é maior que o valor à direita;
- **Menor que (<)**: verifica se o valor à esquerda é menor que o valor à direita;
- **Maior ou igual a (>=)**: verifica se o valor à esquerda é maior ou igual ao valor à direita;

- **Menor ou igual a (\leq)**: verifica se o valor à esquerda é menor ou igual ao valor à direita.

A figura 1 demonstra o funcionamento dos operadores relacionais em Python, utilizando como exemplo as variáveis `a = 10` e `b = 20`. O resultado de cada comparação é exibido como `True` ou `False`, de acordo com a expressão avaliada.

Figura 1 – Exemplo de uso dos operadores relacionais em Python

```
a = 10
b = 20

print(a == b)      # False
print(a != b)     # True
print(a > b)      # False
print(a < b)      # True
print(a >= b)     # False
print(a <= b)     # True
```

Esses operadores são frequentemente utilizados em estruturas condicionais para direcionar o fluxo do programa com base em comparações.

Em Python, as strings (cadeia de caracteres) também podem ser comparadas usando operadores relacionais, mas a semântica difere um pouco da comparação numérica. Para strings, a comparação se dá por ordem lexicográfica (como em um dicionário), levando em conta, por padrão, a ordem dos caracteres no padrão Unicode ou ASCII.

A figura 2 apresenta exemplos de comparação de strings. A comparação lexicográfica verifica o primeiro caractere das duas strings. Se houver diferença, o resultado é decidido imediatamente. Caso sejam iguais, parte para o próximo caractere, e assim sucessivamente.

Figura 2 – Exemplos de comparação de strings

```
# Exemplos de comparação de strings
nome1 = "Ana"
nome2 = "Bruno"
print(nome1 == nome2) # False
print(nome1 != nome2) # True
print(nome1 > nome2) # False (Comparação lexicográfica: "A" < "B")
```

2 Operadores lógicos

Operadores lógicos são utilizados para combinar expressões booleanas, permitindo a construção de condições mais complexas. Em Python, os principais operadores lógicos são:

- **and (E lógico)**: retorna `True` se ambas as expressões forem verdadeiras;
- **or (OU lógico)**: retorna `True` se pelo menos uma das expressões for verdadeira;
- **not (NÃO lógico)**: inverte o valor lógico da expressão; retorna `True` se a expressão for falsa e vice-versa.

A figura 3 demonstra o funcionamento dos operadores lógicos em Python, utilizando como exemplo as variáveis `x = True` e `y = False`. O resultado de cada comparação é exibido como `True` ou `False` de acordo com a expressão avaliada.

Figura 3 – Exemplo de uso de operadores lógicos em Python

```
x = True
y = False

print(x and y) # False
print(x or y) # True
print(not x) # False
```

As tabelas-verdade são representações utilizadas para entender, de forma sistemática, como funcionam os operadores lógicos. Uma tabela-verdade é a representação de todas as combinações possíveis entre os valores de variáveis lógicas em conjunto com operadores lógicos (Forbellone; Eberspächer, 2022). Cada operador lógico (e, ou, não) segue regras próprias para combinar valores booleanos (True ou False), e as tabelas-verdade tornam essas regras claras e sem ambiguidades.

A seguir, apresentamos as tabelas-verdade dos principais operadores lógicos em Python. A tabela 1 mostra o funcionamento do operador `and`, enquanto a tabela 2 exibe os resultados do operador `or`. Já a tabela 3 apresenta a lógica do operador `not`. Esses conceitos são universais, ou seja, não mudam de linguagem para linguagem – apenas a forma de escrever o operador pode variar (por exemplo, em C usamos `&&` para `and`, `||` para `or` e `!` para `not`).

Tabela 1 – Tabela-verdade do operador `and`

EXPRESSÃO A	EXPRESSÃO B	A AND B
False	False	False
False	True	False
True	False	False
True	True	True

Tabela 2 – Tabela-verdade do operador `or`

EXPRESSÃO A	EXPRESSÃO B	A OR B
False	False	False
False	True	True
True	False	True
True	True	True

Tabela 3 – Tabela-verdade do operador not

EXPRESSÃO A	NOT A
False	True
True	False

A figura 4 apresenta exemplos práticos do uso dos operadores relacionais e lógicos em Python. Esses operadores são amplamente utilizados em estruturas de decisão e avaliação de condições. No primeiro exemplo, o operador `and` é utilizado para verificar se uma pessoa tem idade igual ou superior a 18 anos e possui carteira de motorista, permitindo determinar se ela pode dirigir. No segundo exemplo, o operador `or` avalia se é sábado, domingo ou feriado para identificar se o dia é de folga – basta que uma das condições seja verdadeira para que o resultado final também seja. Por fim, o operador `not` é usado para inverter o valor de uma variável booleana, transformando `True` em `False` e vice-versa. Esses exemplos mostram como os operadores lógicos podem ser aplicados para expressar regras e tomar decisões com base em múltiplas condições.

Figura 4 – Exemplos práticos de uso dos operadores relacionais e lógicos

```
# Exemplo 1: Utilizando and
idade = 25
possui_carteira_motorista = True
pode_dirigir = (idade >= 18) and possui_carteira_motorista
# O valor de pode_dirigir será True, pois ambas as condições são verdadeiras

# Exemplo 2: Utilizando or
dia_da_semana = "Sábado"
feriado = False
dia_de_folga = (dia_da_semana == "Sábado") or (dia_da_semana == "Domingo") or feriado
# dia_de_folga será True se for Sábado, Domingo ou se for feriado

# Exemplo 3: Utilizando not
ativado = True
desativado = not ativado # desativado será False
```

3 Expressões lógicas

Uma expressão lógica combina operadores relacionais e lógicos, e seu resultado é sempre um valor booleano (True ou False) (Xavier, 2018). A figura 5 apresenta um exemplo em Python que utiliza `and`, `or` e `not` na mesma expressão.

Ao trabalhar com expressões lógicas, é essencial entender a ordem em que os operadores são avaliados – ou seja, sua precedência. Em Python:

- Os operadores relacionais (`<`, `>`, `==`, `!=`, `<=`, `>=`) são avaliados primeiro.
- O operador `not` tem precedência maior que `and`, que por sua vez tem precedência maior que `or`.
- Parênteses podem (e devem) ser usados para tornar a ordem de avaliação mais explícita – nesse caso, o Python respeita totalmente a hierarquia imposta por eles.

Essa lógica é semelhante à precedência de operadores aritméticos: assim como a multiplicação e a divisão vêm antes da adição e subtração, operadores lógicos também seguem uma ordem definida.

No código da figura 5, vemos uma expressão que envolve todos esses operadores. O passo a passo nos comentários mostra claramente como cada parte da expressão é avaliada até se chegar ao resultado final.

Figura 5 – Exemplo de expressão lógica em Python

```
# Atribuição de valores
x = 10
y = 5
z = 3

# Expressão lógica usando os três operadores (and, or, not)
# Passo a passo:
# 1. (x > y) => 10 > 5 => True
# 2. (y >= z) => 5 >= 3 => True
# 3. Então ((x > y) and (y >= z)) => True and True => True
# 4. A negação not True => False
# 5. (x == z) => (10 == 3) => False
# 6. Conclusão da expressão: False or False => False
resultado = not ((x > y) and (y >= z)) or (x == z)

print(resultado) # False
```

Em programação, é muito comum precisar verificar se um número pertence a um determinado intervalo. Suponha que queremos saber se o valor da variável `n` está entre 10 e 20, incluindo os dois extremos — ou seja, no intervalo [10, 20]. Em Python, podemos construir essa verificação com operadores relacionais combinados com `and`, como mostra o código da figura 6.

Figura 6 – Exemplo de expressão lógica para verificar se um número pertence a um intervalo de valores

```
n = 15
pertence_ao_intervalo = (n >= 10) and (n <= 20)
print(pertence_ao_intervalo) # True
```

Um diferencial interessante de Python é a possibilidade de usar comparações encadeadas que tornam esse tipo de verificação mais concisa e próxima da notação matemática. A figura 7 reescreve o código da figura 6 utilizando comparação encadeada. Esse formato torna o código mais limpo e expressivo, deixando clara a intenção de verificar se um valor está dentro de um intervalo. Além disso, comparações encadeadas

evitam possíveis erros de lógica que podem ocorrer ao usar `and` separadamente.

Comparações encadeadas também são úteis em situações mais complexas. Por exemplo: `10 <= n <= 20 <= m < 100`. Desde que faça sentido no contexto, esse tipo de expressão pode avaliar múltiplas relações de forma clara e eficiente.

Figura 7 – Exemplo de uso de comparação encadeada

```
n = 15
pertence_ao_intervalo = 10 <= n <= 20 # comparação encadeada
print(pertence_ao_intervalo) # True
```

Uma característica importante dos operadores lógicos em programação é o chamado curto-círcuito (*short-circuit*). Esse comportamento significa que, ao avaliar uma expressão lógica, a linguagem interrompe a avaliação assim que encontra um valor suficiente para determinar o resultado final. Por exemplo:

Em `A and B`, se `A` for `False`, o resultado da expressão já será `False`, então `B` não é avaliado.

- Em `A or B`, se `A` for `True`, o resultado será `True`, e `B` também não será avaliado.

A figura 8 apresenta um exemplo de uso de curto-círcuito em expressões lógicas. Neste exemplo, a segunda parte da expressão (`numerador/denominador > 5`) só é avaliada se o denominador for diferente de zero. Isso evita um erro de divisão por zero — algo que aconteceria se a verificação não respeitasse o curto-círcuito.

Figura 8 – Exemplo de uso de curto-círcuito em expressões lógicas

```
numerador = 10
denominador = 0

# Verificamos se a divisão faz sentido antes de realizá-la.
divisao_valida = (denominador != 0) and (numerador / denominador > 5)
```

Além de evitar erros, o curto-círcuito também contribui para a economia de processamento, já que impede a execução de partes desnecessárias do código. Ele torna as expressões mais enxutas e expressivas, permitindo ao programador organizar as condições de forma lógica e segura. Utilizar esse recurso de maneira consciente é uma boa prática de programação, pois além de melhorar o desempenho, reforça a legibilidade e a confiabilidade do código.

Em programação, é comum combinarmos operadores aritméticos com operadores lógicos dentro de uma mesma expressão. Nesse tipo de construção, as operações aritméticas são avaliadas primeiro e, em seguida, seus resultados são comparados por meio de operadores relacionais, gerando um valor booleano (True ou False).

A figura 9 apresenta um exemplo que ilustra exatamente esse processo, detalhando passo a passo como a expressão é avaliada.

Figura 9 – Exemplo de uso de operadores aritméticos em expressões lógicas

```
# Declaração das variáveis
x = 7
y = 4
z = 3

# Passo a passo da avaliação:
# 1) (x + y) = 7 + 4 = 11
# 2) (x + y) > 10 → 11 > 10 → True
#
# 3) (z * 2) = 3 * 2 = 6
# 4) (z * 2) == 6 → 6 == 6 → True
#
# 5) Primeiro resultado parcial: True or True → True
#
# 6) (x - y) = 7 - 4 = 3
# 7) (z + 1) = 3 + 1 = 4
# 8) (x - y) < (z + 1) → 3 < 4 → True
#
# 9) Segundo resultado parcial: True and True → True

resultado = ((x + y) > 10 or (z * 2) == 6) and ((x - y) < (z + 1))

print(resultado) # True
```

O exemplo mostra como expressões compostas podem misturar diferentes tipos de operadores de maneira clara e lógica. Esse tipo de construção é bastante comum ao implementar condições que envolvem cálculos e verificações simultâneas.



PARA SABER MAIS

Para tornar suas expressões lógicas mais claras e alinhadas às melhores práticas da linguagem, recomenda-se a consulta ao guia de estilo do Python, o PEP 8. Esse documento oferece orientações de formatação e convenções que favorecem a legibilidade e a consistência do código. Além disso, ferramentas de análise estática, como Flake8 e pylint, podem identificar pontos de melhoria em expressões complexas e sugerir ajustes que seguem os padrões recomendados.

Considerações finais

Neste capítulo, aprofundamos o entendimento sobre operadores relacionais e lógicos, componentes fundamentais para a construção de decisões em programas. Vimos como esses operadores permitem a comparação de valores e a combinação de condições, além de explorarmos suas aplicações em expressões mais complexas, como o uso de curto-circuito e comparações encadeadas. Embora o foco tenha sido nos operadores em si, é importante destacar que eles são amplamente utilizados em estruturas de controle, como `if` e `while`, que direcionam o fluxo de execução nos programas.

O domínio desses conceitos amplia a capacidade do programador de lidar com cenários reais de forma precisa e otimizada. Seja em estruturas de decisão simples ou na avaliação de múltiplas condições combinadas, os operadores estudados aqui formam a base lógica que sustenta grande parte da programação. Ao aplicar esse conhecimento com consciência e atenção aos detalhes, o desenvolvedor se torna mais preparado para escrever códigos mais expressivos, robustos e alinhados às boas práticas da linguagem Python.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados com aplicações em Python. 4. ed. São Paulo: Pearson; Porto Alegre: Bookman, 2022.

XAVIER, Gley Fabiano Cardoso. **Lógica de programação**. 13. ed. São Paulo: Editora Senac São Paulo, 2018.

Capítulo 7

Estruturas condicionais

Na programação, é comum que um algoritmo precise tomar decisões para adaptar seu comportamento a diferentes situações. Isso significa escolher entre caminhos distintos de execução com base em condições definidas pelo problema. Para implementar essa lógica, utilizamos as chamadas estruturas condicionais – também conhecidas como estruturas de seleção.

Essas estruturas permitem ao programa avaliar expressões lógicas e, com base no resultado, decidir quais blocos de código devem ser executados. Em outras palavras, elas controlam o fluxo do programa de forma dinâmica, tornando-o mais inteligente e adaptável.

Situações típicas em que estruturas condicionais são usadas incluem: verificar dados de login para permitir ou negar acesso a um sistema, identificar se um número é par ou ímpar, avaliar se um aluno atingiu os critérios mínimos para aprovação ou realizar operações bancárias com base no saldo disponível. Em Python, essas decisões são expressas por meio das palavras-chave `if`, `elif` e `else`.

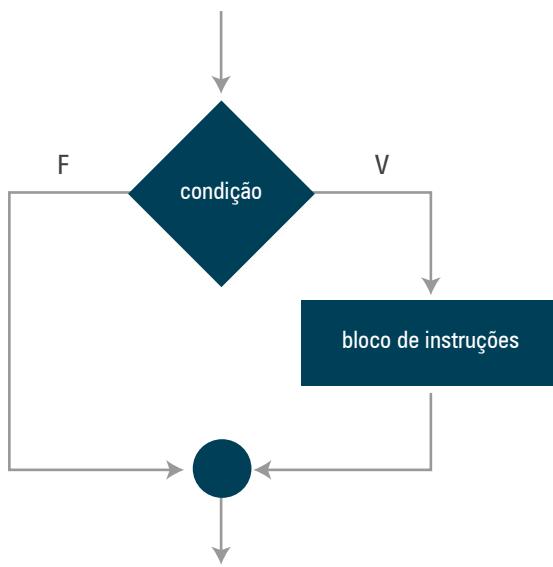
Além de conhecer essas estruturas, é essencial compreender o papel da indentação, que define a hierarquia dos blocos de código em Python. Neste capítulo exploraremos como utilizar as estruturas condicionais de forma eficaz e como organizar logicamente o código para escrever programas claros, corretos e fáceis de manter.

1 Estrutura condicional simples

A estrutura condicional simples é a forma mais básica de decisão em um algoritmo. Ela permite que um determinado bloco de código seja executado apenas se uma condição for verdadeira (Forbellone; Eberspächer, 2022). Caso contrário, nada acontece: o programa simplesmente segue sua execução normalmente, ignorando o bloco condicional.

A figura 1 apresenta a representação da estrutura condicional simples em fluxograma. Nesse tipo de diagrama, o losango é utilizado como símbolo de decisão, representando uma avaliação lógica que determina o fluxo de execução. No interior do losango é colocada uma condição, que será analisada pelo algoritmo. Se a condição for verdadeira (V), o bloco de instruções correspondente é executado. Se a condição for falsa (F), o bloco é ignorado e a execução segue diretamente para a próxima instrução após a estrutura.

Figura 1 – Fluxograma que representa a estrutura condicional simples



Em Python, a estrutura condicional simples é representada pela instrução `if` (se). Esta instrução permite executar um bloco de instruções apenas se uma condição for verdadeira. A figura 2 apresenta a sintaxe da instrução `if`. É importante observar que o bloco de instruções que pertence ao `if` precisa estar indentado (com espaço ou tabulação). Isso é obrigatório, pois a linguagem Python usa indentação para definir blocos de código.

Figura 2 – Sintaxe da instrução if em Python

```
if condição:  
    bloco de instruções
```

Agora que sabemos a sintaxe da instrução `if`, vamos para um exemplo prático que ajuda a entender como aplicá-lo em situações do dia a dia. Imagine um sistema de controle de velocidade em uma via urbana onde o limite máximo permitido é de 50 km/h. Se um carro ultrapassar essa velocidade, o motorista deve ser notificado com uma multa. Essa

é uma situação perfeita para utilizar uma estrutura condicional simples, já que a ação de aplicar a multa só deve acontecer caso a condição (velocidade acima de 50) seja verdadeira. Se a velocidade estiver dentro do limite, o programa simplesmente segue sem executar nenhuma ação. Esse tipo de decisão condicional é a base de muitos sistemas que reagem a dados de entrada e tomam ações de forma automatizada.

Vamos ao enunciado: “Leia a velocidade de um carro. Se ela for superior a 50 km/h, mostre uma mensagem dizendo que o motorista foi multado”. A figura 3 apresenta a solução usando Python. Primeiro, usamos a função `input()` para ler a velocidade informada pelo usuário, convertendo esse valor para número decimal com `float()`. Em seguida, usamos a instrução `if` para verificar se o valor informado ultrapassa o limite. Se a condição for verdadeira, o programa imprime a mensagem.

Figura 3 – Exemplo de uso da estrutura condicional simples

```
velocidade = float(input("Digite a velocidade do carro (em km/h): "))
if velocidade > 50:
    print("Você foi multado por excesso de velocidade!")
```

Esse exemplo mostra como o `if` permite que o programa tome decisões simples com base em condições reais, tornando o código mais inteligente e adaptável às necessidades do mundo real.



IMPORTANTE

Em qualquer estrutura condicional, a condição avaliada pelo programa é sempre interpretada como um valor booleano, ou seja, o resultado será obrigatoriamente True (verdadeiro) ou False (falso). Para construir essas condições, utilizamos dois tipos principais de operadores: os operadores relacionais, que comparam valores, e os operadores lógicos, que permitem combinar ou inverter expressões booleanas. Com a combinação adequada desses operadores é possível representar qualquer lógica de decisão necessária dentro de um programa, desde verificações simples até condições mais complexas.

Podemos também utilizar a estrutura condicional simples para avaliar mais de uma condição ao mesmo tempo. Para isso, usamos expressões lógicas compostas, construídas com os operadores lógicos. Considere, por exemplo, um sistema que permite solicitar um cartão de crédito somente se a pessoa tiver 18 anos ou mais e uma renda mensal mínima de R\$ 2.000,00. Nesse caso, não basta verificar apenas a idade ou apenas a renda: é necessário que ambas as condições sejam verdadeiras. A figura 4 apresenta a implementação desse enunciado em Python.

Figura 4 – Exemplo em Python de múltiplas condições usadas em uma estrutura condicional simples

```
idade = int(input("Digite a idade: "))
renda = float(input("Digite a renda mensal: "))

if idade >= 18 and renda >= 2000:
    print("Pode solicitar o cartão.")
```

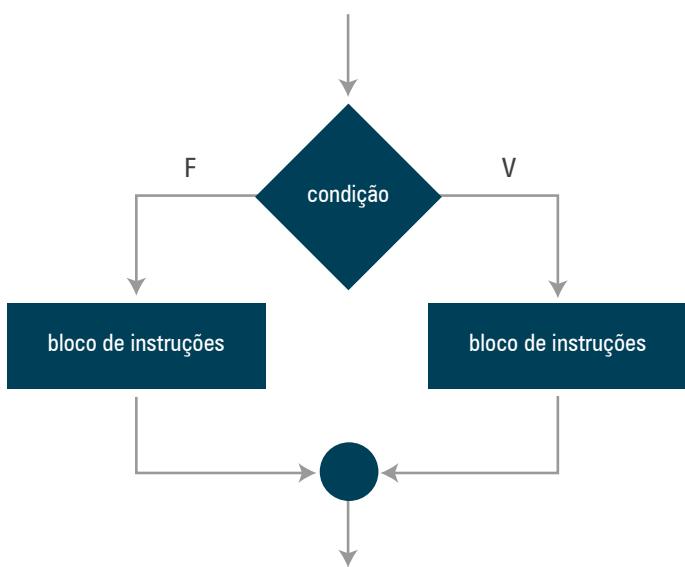
2 Estrutura condicional composta

Agora que você comprehende como a estrutura condicional simples permite executar uma ação somente quando uma condição for verdadeira, vamos dar um passo adiante e explorar a estrutura condicional composta, que amplia essa lógica ao considerar também o que fazer quando a condição for falsa (Forbellone; Eberspächer, 2022). A diferença principal é que, enquanto a estrutura condicional simples só prevê um caminho de execução, a estrutura composta oferece dois caminhos possíveis: um quando a condição for verdadeira e outro quando ela for falsa.

A estrutura condicional composta permite que o programa execute um bloco de instruções quando a condição for verdadeira e outro bloco quando a condição for falsa. É uma extensão natural da estrutura condicional simples, que permite ao programa tomar decisões mais completas, prevendo dois caminhos possíveis.

A figura 5 apresenta a representação da estrutura condicional composta em fluxograma. No interior do losango é colocada uma condição que será avaliada pelo algoritmo. Se a condição for verdadeira (V), o bloco de instruções correspondente é executado. Caso a condição seja falsa (F), é executado um bloco alternativo de instruções. Em ambos os casos, a execução do programa continua normalmente após o término da estrutura.

Figura 5 – Fluxograma representando a estrutura condicional composta



Em Python, a estrutura condicional composta é representada pela instrução `if ... else`. Essa construção permite que o programa execute um bloco de instruções quando uma condição for verdadeira, e outro bloco quando a condição for falsa. A figura 6 apresenta a sintaxe da instrução `if ... else`. Assim como no `if` simples é importante observar que os blocos de comandos que pertencem tanto ao `if` quanto ao `else` devem estar corretamente indentados utilizando espaços ou tabulação.

Figura 6 – Sintaxe da instrução if...else em Python

```
if condição:  
    bloco de instruções  
else:  
    bloco de instruções
```

Vamos voltar ao exemplo do sistema de controle de velocidade em vias urbanas. O limite máximo é de 50 km/h. Se um carro ultrapassar esse limite, o motorista deve ser multado. Mas, se a velocidade estiver dentro do permitido, também podemos mostrar uma mensagem dizendo que está tudo certo. Essa situação é ideal para usar `if ... else`, pois temos dois caminhos possíveis: um para quando o carro passa do limite e outro para quando está dentro da velocidade permitida.

O enunciado é o seguinte: “Leia a velocidade de um carro. Se ela for superior a 50 km/h, mostre uma mensagem dizendo que o motorista foi multado. Caso contrário, informe que a velocidade está dentro do limite permitido”.

A figura 7 apresenta a implementação usando Python. Usamos a função `input()` para ler a velocidade informada pelo usuário, convertendo esse valor para número decimal com `float()`. Em seguida, aplicamos a estrutura `if ... else` para tomar uma decisão com base nesse valor. Se a velocidade for maior que 50 km/h, o programa exibe a mensagem indicando que o motorista foi multado. Caso contrário, ou seja, se a velocidade estiver dentro do limite permitido, o bloco do `else` é executado, exibindo uma mensagem informando que tudo está conforme as regras.

Figura 7 – Exemplo de uso da estrutura condicional composta

```
velocidade = float(input("Digite a velocidade do carro (km/h): "))  
if velocidade > 50:  
    print("Você foi multado por excesso de velocidade.")  
else:  
    print("Velocidade dentro do limite permitido.")
```

Vamos a mais um exemplo prático de uso da estrutura `if...else`, agora aplicando-a a uma situação bastante comum na programação: verificar se um número é par ou ímpar. Sabemos que um número é considerado par quando o resto da divisão por 2 é igual a zero, e ímpar quando esse resto é diferente de zero. Essa verificação pode ser feita de forma simples usando o operador `%` (módulo), que retorna o resto de uma divisão.

O enunciado do problema é: “Leia um número inteiro e informe se ele é par ou ímpar”. A figura 8 apresenta a implementação usando Python. Usamos a função `input()` para ler um valor digitado pelo usuário e convertê-lo para inteiro com `int()`. A seguir, verificamos se o número é divisível por 2 usando `numero % 2 == 0`. Se essa condição for verdadeira, exibimos uma mensagem dizendo que o número é par; caso contrário, a mensagem indica que o número é ímpar.

Figura 8 – Código Python para verificar se um número é par ou ímpar

```
numero = int(input("Digite um número inteiro: "))
if numero % 2 == 0:
    print("O número é par.")
else:
    print("O número é ímpar.")
```

3 Indentação

Em programação, indentação é o deslocamento de uma linha de código para a direita, feito com espaços ou tabulações, para indicar blocos de código ou níveis hierárquicos. Em muitas linguagens de programação, a indentação serve apenas para tornar o código mais legível. No entanto, em Python, ela é obrigatória e faz parte da estrutura sintática da linguagem.

Diferente de linguagens como C, Java ou JavaScript, que utilizam chaves ({}) para delimitar blocos de código, o Python define esses blocos pela indentação. Isso inclui estruturas como `if`, `else`, `elif`, `for`, `while`, funções, entre outros. Isso significa que o nível de indentação informa ao interpretador Python quais instruções pertencem a qual bloco lógico.

A seguir, apresentamos regras básicas de indentação em Python:

1. Cada nível de indentação deve ter a mesma quantidade de espaços;
2. A convenção oficial do Python recomenda o uso de 4 espaços por nível;
3. É possível usar tabulação, mas o ideal é usar espaços (ou configurar o editor para converter tabs em espaços);
4. Não misture espaços e tabulações no mesmo código – isso pode causar erros difíceis de identificar;
5. Após uma instrução que define um bloco (`if`, `else`, `def`, `for`, etc.), a próxima linha deve estar indentada corretamente.

Para indicar que uma instrução não pertence mais a um bloco de código, basta voltar a indentação ao nível anterior. Isso sinaliza ao interpretador que aquele bloco terminou e que uma nova parte do programa será executada fora dele.

A figura 9 apresenta um exemplo de indentação em Python. Neste exemplo, as instruções `print("Parabéns!")` e `print("Você está aprovado.")` fazem parte do bloco `if`, pois estão indentados. Já `print("Fim do programa")` está fora do bloco, pois voltou à margem esquerda.

Figura 9 – Exemplo de indentação em Python

```
nota = 8
if nota >= 7:
    print("Parabéns!")
    print("Você está aprovado.")
print("Fim do programa")
```

Dentro do bloco do if
Dentro do bloco do if
Fora do bloco do if



IMPORTANTE

Em Python, não basta que o código esteja correto logicamente – ele também precisa estar corretamente indentado para funcionar.

4 Estrutura condicional aninhada e encadeada

À medida que os programas se tornam mais complexos, cresce também a necessidade de tomar decisões mais refinadas, que envolvam várias condições. Em situações em que uma única verificação lógica não é suficiente, o programador precisa lidar com múltiplos testes (Guedes, 2014), organizando-os de forma clara e coerente.

Nesse contexto, surgem duas formas muito importantes de estruturar decisões em programas: as estruturas condicionais aninhadas e as estruturas condicionais encadeadas. Essas duas abordagens permitem que o programa:

- Tome decisões em camadas, testando uma condição somente após outra (aninhamento);
- Avalie diversas alternativas mutuamente exclusivas em sequência (encadeamento).

A estrutura condicional aninhada ocorre quando uma condição está dentro de outra. Isso permite criar decisões em etapas, onde a segunda condição só será avaliada se a primeira for verdadeira.

Imagine que estamos desenvolvendo um sistema em que usuários só podem acessar uma determinada função se forem maiores de idade. Mas não basta apenas ter 18 anos ou mais: eles também precisam estar cadastrados como “ativos” no sistema.

Nesse caso, não faz sentido verificar o status de atividade se a idade for menor que 18. Portanto, aplicamos uma decisão aninhada: primeiro testamos a idade; se for válida, então verificamos o status do usuário. A figura 10 apresenta a implementação desta situação em Python. Observe que a verificação do status só acontece se a idade for suficiente, evitando lógica desnecessária. Esse tipo de organização torna o código mais eficiente e mais próximo da lógica real de muitos sistemas.

Figura 10 – Exemplo de uso de estrutura condicional aninhada em Python

```
idade = 20
status = "ativo"
if idade >= 18:
    if status == "ativo":
        print("Acesso liberado.")
    else:
        print("Usuário inativo. Acesso negado.")
else:
    print("Menor de idade. Acesso negado.)
```

A estrutura condicional aninhada é fundamental para lidar com decisões condicionais dependentes. Ela permite que o programa avance em etapas, testando uma condição apenas se outra já tiver sido satisfeita, e assim reflete de forma clara a lógica de muitos problemas do mundo real.

Enquanto a estrutura condicional aninhada é útil para representar decisões hierárquicas em que uma verificação depende logicamente de outra, existem muitos casos em que precisamos apenas escolher entre múltiplas alternativas exclusivas. Nessas situações, utilizamos a estrutura condicional encadeada, que permite ao programa testar uma série de condições diferentes, executando somente o bloco associado à primeira condição verdadeira.

Em Python, essa estrutura é representada pelo uso das seguintes palavras-chave:

- `if` (se)
- `elif` (senão se)
- `else` (senão)

O comando `elif` é uma forma abreviada de “`else if`” e é usado para encadear múltiplas condições dentro de uma estrutura condicional. Ele só é avaliado se a condição anterior (`if` ou outro `elif`) for falsa. A figura 11 apresenta a sintaxe dessa estrutura em Python.

Para usá-la corretamente, é importante entender algumas regras fundamentais de funcionamento do `elif` dentro da estrutura condicional em Python:

- a estrutura começa obrigatoriamente com um `if`;
- você pode ter quantos `elif` quiser, testando diferentes condições;
- o `else` é opcional, e é executado apenas se nenhuma das condições anteriores for satisfeita;
- apenas o primeiro bloco com condição verdadeira será executado.

Figura 11 – Sintaxe da estrutura condicional encadeada em Python

```
if condição1:  
    bloco de instruções 1  
elif condição2:  
    bloco de instruções 2  
elif condição3:  
    bloco de instruções 3  
else:  
    bloco de instruções padrão
```

A estrutura condicional encadeada é especialmente útil quando se quer classificar uma informação em diferentes categorias, aplicar regras alternativas ou definir ações distintas para diferentes valores. Ao usar a combinação `if`, `elif` e `else`, o código se torna mais limpo, organizado e evita a redundância de múltiplos `if` isolados que poderiam gerar resultados contraditórios ou múltiplas saídas inesperadas.

Por exemplo, imagine um sistema educacional que atribui conceitos ao desempenho dos alunos com base em faixas de nota. Se a nota for maior ou igual a 9, o desempenho é considerado excelente. Se for entre 7 e 9, é bom. Entre 5 e 7, regular. Abaixo de 5, insuficiente.

Esse é um caso clássico de uso da estrutura condicional encadeada, pois a nota pertence a uma e apenas uma das faixas. Assim, o programa deve parar a verificação na primeira condição satisfeita. A figura 12 apresenta a implementação dessa situação em Python.

Figura 12 – Exemplo de uso de estrutura condicional encadeada em Python

```
nota = float(input("Digite a nota final do aluno: "))
if nota >= 9:
    print("Desempenho: Excelente")
elif nota >= 7:
    print("Desempenho: Bom")
elif nota >= 5:
    print("Desempenho: Regular")
else:
    print("Desempenho: Insuficiente")
```

Perceba que a ordem das condições é importante. Ao testar da maior faixa para a menor, garantimos que o aluno receba o conceito mais elevado possível de acordo com a nota. Isso também evita que mais de um bloco seja executado, pois o Python interrompe a cadeia de verificações assim que encontra a primeira condição verdadeira.

A estrutura condicional encadeada é uma das formas mais eficientes e organizadas de lidar com decisões múltiplas e exclusivas em um programa. Ela permite ao código seguir um fluxo lógico único, escolhendo uma entre várias possibilidades, sem repetições ou ambiguidade. Ao dominar seu uso, o programador é capaz de expressar decisões mais complexas de forma simples, clara e alinhada com a lógica dos problemas do mundo real.

Considerações finais

Neste capítulo, exploramos as estruturas condicionais fundamentais para o desenvolvimento de algoritmos mais inteligentes e adaptáveis. Iniciamos com a estrutura condicional simples, que permite a execução de um bloco de código somente quando uma condição é verdadeira. Em seguida, avançamos para a estrutura composta, que contempla alternativas para ambos os cenários — verdadeiro e falso —, ampliando a capacidade de resposta dos programas às diversas situações do mundo real.

Também discutimos a importância da indentação em Python, não apenas como uma convenção de estilo, mas como um elemento essencial da sintaxe da linguagem. Além disso, aprendemos a lidar com decisões mais complexas por meio de estruturas aninhadas e encadeadas que possibilitam uma lógica mais refinada, especialmente em sistemas que exigem múltiplas verificações ou classificações por categorias.

Ao dominar as estruturas condicionais, o programador passa a ter controle mais preciso sobre o fluxo de execução do seu código, tornando-o mais claro, eficiente e coerente com a lógica dos problemas que busca resolver. Fica como reflexão a importância de escrever decisões bem estruturadas, pensando não apenas na funcionalidade, mas também na legibilidade e manutenção futura do código.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico.

Lógica de programação: a construção de algoritmos e estruturas de dados com aplicações em Python. 4. ed. São Paulo: Pearson; Porto Alegre: Bookman, 2022.

GUEDES, Sérgio. **Lógica de programação algorítmica.** São Paulo: Pearson, 2014. *E-book*.

Capítulo 8

Estruturas de repetição

Imagine que você precise realizar uma tarefa não uma, mas dez, cem, ou talvez milhares de vezes. Seja calcular a média de notas de todos os alunos de uma universidade, processar cada linha de um arquivo de dados gigantesco ou simplesmente exibir uma mensagem na tela repetidamente. Realizar essas tarefas manualmente seria não apenas tedioso e propenso a erros, mas, em muitos casos, humanamente impossível dentro de um prazo razoável.

Na programação, frequentemente surgem situações em que é necessário executar um determinado bloco de instruções repetidamente. Exemplos incluem cálculos de médias, validação contínua de entradas de usuários, processamento sistemático de dados e iterações sobre coleções. Para solucionar tais problemas com eficiência e clareza, as linguagens de programação fornecem recursos específicos denominados estruturas de repetição (Forbellone; Eberspächer, 2022). Este capítulo explora detalhadamente duas estruturas fundamentais em Python: `while` e `for`, esclarecendo seu funcionamento, aplicações práticas e diferenças essenciais.

Estruturas de repetição, também conhecidas como laços (ou *loops*), permitem a execução repetida de um conjunto de instruções enquanto uma determinada condição se mantiver verdadeira. Ao usar esses recursos, os programas tornam-se mais compactos e claros, evitando redundâncias e facilitando a manutenção do código.

Essas estruturas geralmente são classificadas em dois tipos principais:

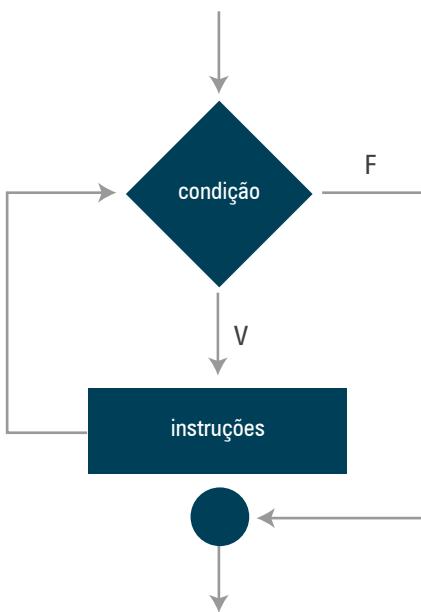
- Repetição condicional (`while`): executa repetidamente instruções enquanto uma condição específica permanecer verdadeira.
- Repetição por contagem ou iteração (`for`): percorre uma sequência predeterminada ou conhecida para realizar a repetição.

Nas próximas seções, apresentaremos cada uma dessas estruturas, destacando suas funcionalidades específicas, usos recomendados e características fundamentais para que você possa aplicá-las corretamente em suas soluções de programação.

1 Estrutura de repetição `while`

A estrutura de repetição `while` (enquanto) permite que um bloco de instruções seja executado repetidamente enquanto uma condição booleana permanecer verdadeira (Forbellone; Eberspächer, 2022). Esse laço atua como uma verificação condicional constante, avaliando a condição antes de cada nova execução do bloco. Se a condição for avaliada como verdadeira (V), o bloco de instruções é executado; caso seja falsa (F), o laço é encerrado e a execução do programa segue adiante. Após cada execução do bloco, a condição é novamente verificada, podendo assim permitir novas iterações. A figura 1 apresenta a representação dessa estrutura de repetição em forma de fluxograma.

Figura 1 – Fluxograma representando a estrutura de repetição while



Para entender claramente o funcionamento do laço `while`, considere o seguinte exemplo cotidiano: imagine que você esteja enchendo um copo com água. Você continua enchendo o copo enquanto ele não estiver cheio. Nesse caso, a condição “o copo não está cheio” é testada continuamente antes de adicionar mais água.

De forma análoga, no contexto da programação, o bloco de instruções do `while` é repetido enquanto uma condição semelhante (algo ainda não atingido ou não realizado) estiver verdadeira.



IMPORTANTE

O laço `while` é utilizado quando não sabemos exatamente quantas vezes uma repetição deve ocorrer, mas temos uma condição que deve ser satisfeita para que a repetição continue.

A figura 2 apresenta a sintaxe geral da instrução `while` em Python. O fluxo de execução da estrutura `while` inicia com a avaliação da condição estabelecida. Se essa condição for avaliada como verdadeira (`True`), o bloco de código associado é executado. Após essa execução, a condição é verificada novamente. Esse ciclo se repete enquanto a condição continuar verdadeira. Quando a condição for avaliada como falsa (`False`), a repetição é interrompida e a execução do programa prossegue a partir da linha imediatamente após o laço.

Figura 2 – Sintaxe da instrução while em Python

```
while condição:  
    bloco de instruções
```

Considere um programa que imprime na tela os números de 1 a 5. Essa tarefa representa um exemplo clássico para ilustrar o uso da estrutura de repetição `while`, já que envolve a execução repetida de uma instrução por um número definido de vezes. A lógica da solução baseia-se no uso de uma condição de parada, responsável por controlar quantas vezes o bloco de comandos será executado. O programa inicia com a definição de uma variável chamada `contador`, que recebe um valor inicial, e, a cada repetição, tem seu valor atualizado de forma a garantir que, em algum momento, a condição se torne falsa e o laço seja finalizado. A figura 3 apresenta a implementação desse programa em Python.

Figura 3 – Código em Python para imprimir os números de 1 a 5 na tela

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

Um elemento central nesse tipo de estrutura é a **variável contadora** – uma variável cuja função é registrar e controlar o número de vezes que

o laço é executado. Essa variável é geralmente inicializada com um valor antes do início do laço e, dentro do bloco de repetição, é atualizada a cada ciclo, normalmente por meio de incrementos. A variável contadora atua, portanto, como um mecanismo de controle que permite limitar a quantidade de repetições de maneira previsível.

Além disso, o uso adequado da variável contadora ajuda a tornar o código mais claro, seguro e eficiente. Ela permite que o programador acompanhe o progresso do laço e crie condições específicas de parada, possibilitando maior controle lógico sobre o comportamento do programa.

Considere agora o cenário de lançamento de um foguete. Normalmente ocorre uma contagem regressiva até chegar a zero, momento em que o foguete decola. Vamos simular essa contagem regressiva usando um laço `while`. A figura 4 apresenta a implementação deste cenário em Python.

Figura 4 – Implementação em Python de um cenário de contagem regressiva

```
contagem = 10
while contagem > 0:
    print(f"Faltam {contagem} segundos para o lançamento.")
    contagem = contagem - 1
print("Lançamento!")
```

Outra aplicação bastante comum das estruturas de repetição é o cálculo de somatórios. Nesse tipo de problema, desejamos somar uma sequência de valores que seguem um padrão definido, o que torna o uso de laços ideal para a tarefa. A figura 5 apresenta o código de um programa que calcula a soma dos números inteiros de 1 a 6 utilizando a estrutura `while`.

Figura 5 – Código de um programa que calcula a soma dos números inteiros de 1 a 6

```
numero = 1
soma = 0
while numero <= 6:
    soma += numero
    numero += 1
print("Somatório de 1 a 6:", soma)
```

A lógica do programa é: começamos com duas variáveis – `numero`, que será usada como contador, e `soma`, que armazenará o valor acumulado da soma. A cada repetição do laço, o valor atual de `numero` é somado ao total já armazenado na variável `soma`, e em seguida `numero` é incrementado. O laço continua até que `numero` ultrapasse 6. Ao final da repetição, a variável `soma` conterá o resultado final da operação: 21.

Nesse contexto, utilizamos um tipo especial de variável chamada **variável acumuladora**. Esse tipo de variável tem como função acumular valores ao longo das iterações de um laço, permitindo a construção progressiva de um resultado final. No exemplo apresentado, `soma` é a variável acumuladora, pois a cada repetição ela armazena o valor anterior somado ao novo valor de `numero`.

É importante destacar que, para funcionar corretamente, a variável acumuladora precisa ser inicializada com um valor adequado antes do início da repetição – no caso de somatórios, esse valor geralmente é zero. A cada ciclo do laço, o novo valor é agregado ao valor já existente, formando assim uma sequência de atualizações que leva ao resultado esperado. Esse tipo de variável é amplamente utilizado em algoritmos que envolvem cálculos de totais, médias, contagens de eventos e outras formas de processamento acumulativo de dados.

Em muitas aplicações, no entanto, não é possível saber com antecedência quantas vezes o laço será executado. Isso ocorre, por exemplo, em programas que interagem com o usuário, nos quais a quantidade de

repetições depende de valores fornecidos durante a execução. Nessas situações, a estrutura de repetição `while` se mostra especialmente adequada, pois permite que o laço continue funcionando até que uma condição definida em tempo de execução seja satisfeita, oferecendo flexibilidade para lidar com entradas dinâmicas e cenários imprevisíveis.

Considere a seguinte situação: queremos criar um programa que leia números inteiros positivos (incluindo o zero) informados pelo usuário. A entrada de dados deve continuar até que o usuário digite um número negativo, que nesse caso atua como um valor sentinelas — um valor especial utilizado para indicar o fim da entrada de dados. Ao final, o programa deve exibir a soma e a média de todos os números positivos digitados. A figura 6 apresenta a implementação desse cenário em Python.

Neste código, o laço `while` continua executando enquanto o usuário digitar valores maiores ou iguais a zero. A cada entrada válida, o número é somado à variável `soma` e o contador `quantidade` é incrementado. Quando o número negativo (valor sentinelas) é digitado, o laço é interrompido. Em seguida, o programa verifica se houve pelo menos uma entrada válida e, se sim, calcula e exibe a média.

Esse tipo de situação ilustra bem o uso de *laços com condição de parada indefinida*, em que o número de repetições não é conhecido antecipadamente, mas é determinado pelo comportamento do usuário ou de outra fonte de dados externa.

Observe que, neste programa, utilizamos duas variáveis fundamentais no contexto das estruturas de repetição: uma acumuladora e uma contadora. A variável `soma` atua como acumuladora, pois armazena progressivamente os valores somados ao longo das iterações. Já a variável `quantidade` funciona como contadora, registrando quantas vezes o laço executou uma operação válida.

O uso combinado de variáveis acumuladoras e contadoras é bastante comum em algoritmos que lidam com entradas repetidas, permitindo realizar operações como o cálculo de totais, médias, frequências e outros resultados derivados de uma sequência de dados. Compreender essa lógica é essencial para o desenvolvimento de programas mais robustos, interativos e preparados para processar dados fornecidos de forma dinâmica.

Figura 6 – Exemplo de estrutura de repetição com condição de parada indefinida

```
numero = 0
soma = 0
quantidade = 0

while numero >= 0:
    numero = int(input("Digite um número positivo (ou negativo para encerrar): "))

    if numero >= 0:
        soma += numero
        quantidade += 1

    if quantidade > 0:
        media = soma / quantidade
        print("Soma dos números:", soma)
        print("Média dos números:", media)
    else:
        print("Nenhum número positivo foi digitado.")
```

Uma preocupação comum ao utilizar estruturas `while` é garantir que a condição de parada possa realmente acontecer. Caso contrário, o programa pode entrar em um loop infinito, executando os comandos continuamente sem parar. A figura 7 apresenta um exemplo de um laço infinito.

Figura 7 – Exemplo de um laço infinito

```
# Loop infinito - não execute!
while True:
    print("Loop infinito")
```

Para evitar loops infinitos, sempre se assegure de que exista alguma instrução dentro do bloco que torne a condição falsa em algum momento.

2 Estrutura de repetição `for`

Em muitas situações na programação, precisamos repetir uma determinada ação um número conhecido e controlado de vezes. Seja para imprimir uma sequência de números, realizar uma determinada operação para cada item de um conjunto de valores, ou simplesmente executar um bloco de código um número fixo de vezes, é comum nos depararmos com repetições previsíveis e bem definidas. Para esses casos, a linguagem Python oferece a estrutura de repetição `for` (para), que permite percorrer sequências de forma eficiente, legível e controlada.

Diferente do laço `while`, que é mais indicado para repetições de natureza indefinida (em que o número de execuções depende de uma condição externa), o laço `for` é ideal quando sabemos de antemão quantas vezes a repetição deve ocorrer, ou quando queremos iterar sobre elementos de uma sequência, como números em um intervalo.

O laço `for` permite percorrer uma sequência de valores, executando um bloco de código para cada valor dessa sequência. Em Python, a forma mais comum de uso do `for` é com a função `range()`, que gera uma sequência numérica que pode ser usada para controlar as repetições (Silva; Fortes, 2022).

A figura 8 apresenta a sintaxe geral da instrução `for` na linguagem Python. Onde:

- `início`: valor inicial da sequência (opcional, padrão é 0);
- `fim`: valor final da sequência (não incluído);
- `passo`: valor do incremento entre as repetições (opcional, padrão é 1).

Figura 8 – Sintaxe da instrução for em Python

```
for variavel in range(início, fim, passo):
    bloco de instruções
```

Durante a execução, a variável definida no for assume, a cada iteração, um valor da sequência gerada por `range()`. Isso permite controlar com precisão quantas vezes o bloco será repetido e, se necessário, utilizar esse valor dentro do próprio bloco de comandos.

Vamos implementar novamente um programa que imprime na tela os números de 1 a 5, agora usando o laço **for**. A figura 9 apresenta o código do programa em Python.

Esse código imprime os números de 1 a 5, um por linha. A função `range(1, 6)` gera os valores 1, 2, 3, 4 e 5 – o valor final (6) não é incluído na sequência. A variável `numero` assume cada um desses valores a cada repetição do laço.

Figura 9 – Código em Python que imprime os números de 1 a 5 na tela utilizando for

```
for numero in range(1, 6):
    print(numero)
```

A função `range()` é uma das principais ferramentas associadas ao laço for em Python. Ela permite gerar uma sequência de números inteiros, que pode ser usada para controlar a quantidade de repetições de forma previsível. O `range()` pode ser utilizado de três maneiras diferentes, dependendo da quantidade de argumentos fornecidos: com um único valor (`fim`), com dois valores (`início` e `fim`), ou com três valores (`início`, `fim` e `passo`). A figura 10 apresenta um exemplo de código para cada uma dessas variações.

Figura 10 – Exemplos de uso da função range()

```
# Contagem simples: apenas o valor final
# Apresenta os números de 0 a 4
for i in range(5):
    print("Número:", i)

# Contagem com valor inicial e final
# Apresenta os números de 1 a 9
for i in range(1, 10):
    print("Número:", i)

# Contagem com passo definido
# Apresenta os números: 0, 2, 4, 6, 8, 10
for i in range(0, 11, 2):
    print("Número:", i)
```

Outra aplicação do parâmetro de passo da função `range()` é a contagem decrescente, que pode ser feita ao definir um passo negativo. Para isso, o valor inicial deve ser maior que o valor final. A figura 11 apresenta um código com exemplo de contagem regressiva.

Figura 11 – Exemplo de código para contagem regressiva usando a instrução for

```
for i in range(5, 0, -1):
    print("Contagem regressiva:", i)
```

A saída será: 5, 4, 3, 2, 1. Esse tipo de laço é bastante utilizado em situações como cronômetros, processos de finalização ou mensagens temporizadas. É importante lembrar que, nesse caso, o valor final também não é incluído.

Vamos a um outro exemplo do uso da instrução `for`. Imagine que o programa deve apresentar a tabuada de multiplicação de um número informado pelo usuário. Nesse caso, sabemos exatamente quantas vezes o cálculo deverá ser realizado: de 1 a 10. Esse tipo de repetição é ideal para ser resolvido com a estrutura `for`, pois a quantidade de iterações é conhecida previamente, o que facilita o controle do laço. A figura 12 apresenta a implementação em Python deste programa.

Figura 12 – Código para apresentar a tabuada de um número informado pelo usuário

```
numero = int(input("Digite um número para ver a tabuada: "))
print(f"Tabuada de {numero}:")
for i in range(1, 11):
    resultado = numero * i
    print(f"{numero} x {i} = {resultado}")
```

Neste exemplo, o usuário digita um número e o programa imprime a tabuada correspondente, multiplicando esse número por valores de 1 a 10. A função `range(1, 11)` gera os números de 1 até 10 (lembrando que o valor final não é incluído), e cada valor é usado na multiplicação.

Uma das principais vantagens do `for` em Python é sua clareza e simplicidade. O programador não precisa se preocupar em atualizar manualmente o valor da variável de controle nem em verificar a condição de parada — tudo isso é gerenciado automaticamente pela estrutura do laço. Isso torna o código mais limpo, seguro e fácil de manter.

Enquanto o laço `while` é mais adequado para casos em que a condição de repetição depende de fatores externos e incertos (como a entrada do usuário), o `for` se destaca quando o controle da repetição pode ser determinado antecipadamente.

3 Controle de fluxo com as instruções `break` e `continue`

Durante a execução de um laço de repetição, pode haver situações em que seja necessário interromper a repetição antes do previsto ou pular uma determinada iteração sem encerrar o laço por completo. Para esses casos, a linguagem Python oferece duas instruções muito úteis: `break` e `continue`. Ambas permitem controlar o fluxo interno dos laços (`while` ou `for`), oferecendo mais flexibilidade ao programador.

A instrução `break` serve para encerrar imediatamente a execução de um laço, mesmo que a condição de repetição ainda seja verdadeira. Isso é útil, por exemplo, quando o programa encontra um resultado esperado ou detecta uma condição que torna desnecessário continuar repetindo o bloco de código. A figura 13 apresenta um exemplo de uso da instrução `break`.

Figura 13 – Exemplo de uso da instrução break

```
while True:  
    numero = int(input("Digite um número (0 para sair): "))  
    if numero == 0:  
        print("Encerrando o programa.")  
        break  
    print("Você digitou:", numero)
```

Nesse caso, o laço `while` está configurado para ser infinito (`while True`). No entanto, quando o usuário digita o número 0, a instrução `break` é acionada e o laço é encerrado imediatamente. Esse padrão é muito comum em programas interativos, especialmente em menus, jogos ou sistemas que aguardam comandos do usuário.

Já a instrução `continue` tem a função de interromper apenas a iteração atual do laço, fazendo com que o fluxo de execução retorne ao início do laço para avaliar novamente a condição de repetição (no caso do `while`) ou passar para o próximo valor (no caso do `for`). A figura 14 apresenta um exemplo de uso da instrução `continue`, que imprime os números ímpares entre 1 e 10.

Figura 14 – Exemplo de uso da instrução continue

```
for i in range(1, 11):  
    if i % 2 == 0:  
        continue  
    print("Número ímpar:", i)
```

Neste código, sempre que `i` for um número par (`i % 2 == 0`), a instrução `continue` será acionada e o `print()` não será executado para aquela iteração. Assim, somente os números ímpares são exibidos. Esse tipo de lógica é útil quando determinadas condições devem ser ignoradas durante a repetição, mas sem encerrar o laço como um todo.

Embora essas instruções sejam úteis, é importante utilizá-las com cautela. Um uso excessivo ou desorganizado de `break` e `continue` pode dificultar a leitura e a manutenção do código. Sempre que possível, opte por condições bem definidas no controle do laço e utilize essas instruções como ferramentas complementares quando realmente forem necessárias para resolver um problema de lógica.

Considerações finais

Neste capítulo, exploramos as estruturas de repetição em Python com foco nos laços `while` e `for`, além das instruções de controle `break` e `continue`. Compreender esses recursos é essencial para desenvolver programas que realizem tarefas de forma repetitiva, de maneira eficiente e estruturada. Através de exemplos práticos, foi possível observar como essas estruturas permitem controlar o fluxo de execução com base em condições ou em sequências definidas, tornando os algoritmos mais compactos, dinâmicos e fáceis de manter.

Destacamos a importância do uso consciente de variáveis contadoras e acumuladoras, que são fundamentais no controle lógico de laços e na construção de soluções que processam séries de dados. Também vimos que a escolha entre `while` e `for` depende do tipo de repetição desejada: condicional ou por contagem. Além disso, as instruções `break` e `continue` se mostraram úteis para situações que exigem controle mais refinado dentro dos laços, embora devam ser utilizadas com moderação para garantir a clareza do código.

Por fim, vale refletir sobre como o domínio dessas estruturas básicas representa um passo fundamental no processo de aprendizado da programação. Com elas, o programador é capaz de automatizar tarefas, validar entradas, realizar cálculos em série e muito mais. Ao praticar os conceitos apresentados, você desenvolverá a base necessária para enfrentar desafios mais complexos, com algoritmos mais eficientes e soluções mais criativas.

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados com aplicações em Python. 4. ed. São Paulo: Bookman, 2022.

SILVA, Leonardo Soares e; FORTES, Gabriel. **Aprenda a programar com Python**: descomplicando o desenvolvimento de software. São Paulo: Casa do Código, 2022.

Sobre o autor

Alexandre dos Santos Mignon é bacharel em ciência da computação pela PUC-SP (2001), especialista em tecnologia de objetos pelo Senac-SP (2003), mestre e doutor em engenharia da computação pela Poli-USP.

Na área educacional, possui mais de 15 anos de experiência como professor de disciplinas de programação e engenharia de software.

