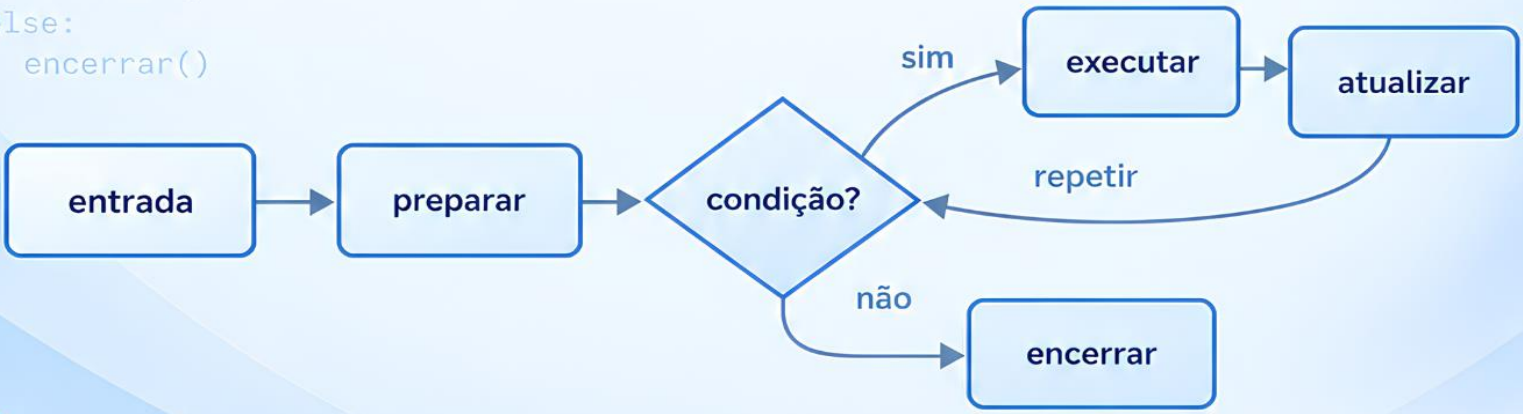
Jose Augusto dos Santos Silva

Lógica de Programação em Python:

Fundamentos, Estruturas de Controle e Resolução de Problemas

```
def main():
  entrada = input()
  if condicao:
     executar()
  else:
     encerrar()
```

```
while True:
  atualizar()
  if fim:
    break
```



thesis editora científica

Lógica de Programação em Python

Fundamentos, Estruturas de Controle e Resolução de Problemas

Jose Augusto dos Santos Silva

Engenheiro de Computação pelo Instituto de Computação da Universidade Federal de Alagoas
Mestre em Informática pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas

Livro didático de Ciência da Computação

2026

2026 - Thesis Editora Científica

Editor Chefe: Felipe Cardoso Rodrigues Vieira

Diagramação, Projeto Gráfico e Design da Capa: Thesis Ed. Cien. e Jose Augusto dos Santos Silva

Revisão: Jose Augusto dos Santos Silva

O manuscrito foi previamente submetido à avaliação cega pelos pares (*blind peer review*), membros do Conselho Editorial desta Editora, tendo sido aprovado para a publicação com base em critérios de neutralidade e imparcialidade acadêmica.

ISBN: 978-65-83199-78-2

Thesis Editora Científica

Teresina – PI – Brasil

contato@thesiseditora.com.br

www.thesiseditora.com.br

Conselho editorial

2026 - Thesis Editora Científica

Editor Chefe: Felipe Cardoso Rodrigues Vieira

Diagramação, Projeto Gráfico e Design da Capa: Thesis Ed. Cien. e Jose Augusto dos Santos Silva

Revisão: Jose Augusto dos Santos Silva

Catalogação na Publicação

Elaborada por Bibliotecária Janaina Ramos – CRB-8/9166

S586L

Silva, Jose Augusto dos Santos

Lógica de programação em Python: fundamentos, estruturas de controle e resolução de problemas / Jose Augusto dos Santos Silva. – Teresina-PI: Thesis Editora Científica, 2026.

Livro em PDF

ISBN 978-65-83199-78-2

1. Programação de computadores. 2. Python (Linguagem de programação). I. Silva, Jose Augusto dos Santos. II. Título.

CDD 005.1

Índice para catálogo sistemático

I. Programação de computadores

Thesis Editora Científica

Teresina – PI – Brasil

contato@thesiseditora.com.br

www.thesiseditora.com.br

Dados editoriais

Esta obra foi organizada como livro didático para estudantes de Ciência da Computação, com explicações graduais, exemplos executáveis, atividades de fixação e referências técnicas.

Jose Augusto dos Santos Silva é Engenheiro de Computação pelo Instituto de Computação da Universidade Federal de Alagoas. Também é Mestre em Informática pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Os códigos foram escritos para fins didáticos. O leitor deve executá-los em ambiente Python compatível e adaptar exemplos, nomes e dados ao contexto de seus próprios estudos.

Nota de responsabilidade. Conceitos, exemplos e exercícios devem ser lidos como material educacional. A execução de programas em sistemas reais exige avaliação de segurança, desempenho e manutenção.

Apresentação

Aprender Computação envolve mais do que aprender uma linguagem. É necessário representar problemas, separar decisões, escolher estruturas adequadas, testar hipóteses e explicar por que uma solução funciona. Este livro foi escrito para acompanhar esse percurso em passos curtos, com exemplos que podem ser lidos, executados, modificados e discutidos.

Python é utilizada como ferramenta de expressão porque permite concentrar a atenção na lógica. A linguagem não elimina a necessidade de raciocínio: ela apenas reduz o ruído sintático para que o estudante observe dados, estados, condições, repetições e funções.

O texto alterna explicações, tabelas, diagramas, pequenos programas e exercícios. A recomendação é não tratar os exemplos como respostas para copiar. Primeiro leia o problema, formule uma previsão, escreva uma solução mínima e só então compare o resultado com o programa apresentado.

Como usar este livro

- Leia a explicação antes de executar o código; identifique entradas, processamento e saídas.
- Digite ou reescreva os exemplos, pois a formulação manual revela detalhes que a leitura passiva esconde.
- Altere um elemento por vez e observe como a execução muda.
- Resolva os exercícios sem consultar a resposta orientativa; use a resposta para revisar o raciocínio.
- Mantenha um diário de erros com a mensagem recebida, a hipótese testada e a correção realizada.

Sumário

Os títulos abaixo possuem navegação interna no arquivo. Clique em qualquer item para ir diretamente à seção.

Apresentação 7
Como usar este livro 8
Pensar antes de programar 12
Primeiros passos com Python 16
Dados e expressões 19
Decisões e caminhos 22
Repetições e padrões 26
Funções e modularização 30
Textos e coleções simples 33
Arquivos, exceções e depuração 36
Projetos integradores 40
Revisão geral e próximos passos 43
Caderno de laboratório 44
Banco de desafios e revisão 57
Guia de estudo e revisão 59
Projeto guiado organizador de estudos 67
Respostas orientativas 72
Glossário essencial 73
Referências 74
Sobre o autor 75

Lista de figuras

As figuras são numeradas na ordem em que aparecem e possuem navegação interna no arquivo.

Figura 1 — O ciclo de construção de uma solução computacional 13
Figura 2 — O caminho de um arquivo Python até a saída 16
Figura 3 — Uma decisão com caminhos alternativos 22
Figura 4 — Etapas de um ciclo while 27
Figura 5 — Decomposição de um programa em funções 30
Figura 6 — Ciclo de depuração 38
Figura 7 — Caminhos de um algoritmo 60

Lista de tabelas

As tabelas são numeradas na ordem em que aparecem e possuem navegação interna no arquivo.

Tabela 1 — Representações úteis em cada momento 13
Tabela 2 — Perguntas para ler um programa 17
Tabela 3 — Tipos comuns em programas introdutórios 19
Tabela 4 — Estratégias para organizar decisões 23
Tabela 5 — Padrões de estado em repetições 28
Tabela 6 — Função mais fácil de testar 31
Tabela 7 — Qual coleção comunica melhor a intenção? 34
Tabela 8 — Possível organização do projeto 41
Tabela 9 — Lista de verificação do projeto 42
Tabela 10 — Exemplo de faixas 46
Tabela 11 — Pergunta e coleção 51
Tabela 12 — Roteiro de entrega 55
Tabela 13 — Rubrica 58
Tabela 14 — Rastreamento de uma soma 61
Tabela 15 — Categorias de teste 62
Tabela 16 — Padrões de crescimento 64
Tabela 17 — Ciclo semanal 65
Tabela 18 — Contrato do organizador 67
Tabela 19 — Casos de validação 69
Tabela 20 — Checklist da entrega 71

CAPÍTULO 1

Pensar antes de programar

Um programa é uma explicação executável de uma ideia.

Computação como resolução de problemas

Objetivos de aprendizagem. Ao concluir esta parte, você deverá conseguir:

- distinguir problema, modelo, algoritmo e programa;
- identificar entradas, processamento e saídas;
- decompor uma tarefa em partes menores e verificáveis.

Programar é construir uma descrição precisa de um processo. Antes de escolher comandos, é preciso compreender o problema: o que está sendo solicitado, quais dados estão disponíveis, quais regras devem ser respeitadas e qual resultado será considerado correto. Essa etapa parece simples, mas é responsável por grande parte dos erros de projetos pequenos. Quando uma pessoa começa a escrever código sem delimitar a tarefa, tende a misturar entrada, cálculo, apresentação e decisões em um único bloco difícil de testar.

Um problema computacional pode ser visto como uma transformação. Há uma entrada, existe um processamento e espera-se uma saída. A mesma ideia vale para um programa que calcula uma média, para uma rotina que organiza nomes ou para um sistema que consulta dados. A fórmula não precisa ser numérica: transformar uma lista de pedidos em uma fila de atendimento também é uma operação computacional. Pensar nessa transformação ajuda a separar o que é regra do domínio e o que é detalhe da linguagem.

A decomposição torna o problema manejável. Em vez de tentar resolver um sistema de biblioteca de uma só vez, podemos separar cadastro, empréstimo, devolução, busca e relatório. Cada parte ainda pode ser dividida em tarefas menores. Uma decomposição útil vai além de listar telas: ela separa responsabilidades que podem ser explicadas e verificadas isoladamente.

Não é preciso antecipar todos os detalhes. A decomposição reduz a quantidade de decisões simultâneas. Pergunte se cada tarefa pode ser descrita em poucas frases, se possui entradas e saídas claras e se pode ser testada sem depender do programa inteiro. Quando essas respostas aparecem, a implementação fica mais previsível.

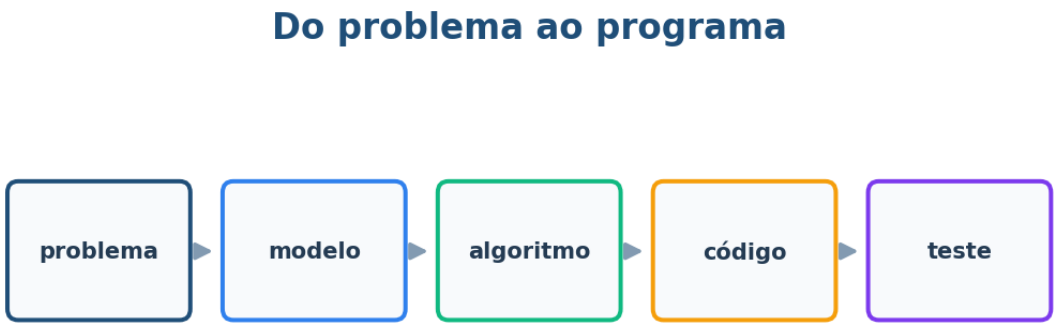


Figura 1 — O ciclo de construção de uma solução computacional

Síntese do capítulo

- A programação começa pela compreensão do problema, não pela escolha de um comando.
- Entrada, processamento e saída formam um modelo simples para organizar a descrição.
- Decompor é transformar uma tarefa grande em responsabilidades menores e testáveis.

Algoritmos e representações

Um algoritmo é uma sequência finita de passos que orienta a solução de um problema. A palavra sequência não significa que todos os passos serão executados sempre: um algoritmo pode conter decisões e repetições. O que importa é que as regras de execução sejam suficientemente claras para que outra pessoa consiga segui-las e obter o mesmo resultado.

A linguagem natural é útil para conversar sobre a solução, mas pode ser ambígua. Expressões como faça rapidamente ou se necessário, corrija o valor não definem operações observáveis. O pseudocódigo ocupa um meio-termo: mantém vocabulário próximo do problema, mas explicita atribuições, condições, repetições e retornos. O fluxograma oferece outra perspectiva, mostrando o caminho de controle por meio de símbolos e setas.

Não existe uma única representação correta para todo problema. Uma lista de passos é adequada para uma rotina curta; uma tabela de rastreamento é útil quando o valor de uma variável muda em várias etapas; um fluxograma ajuda a discutir caminhos alternativos; e o código é a forma executável. A escolha deve ser orientada pela pergunta que se quer responder.

Um algoritmo também precisa de critérios de término. Uma repetição sem condição de saída não é uma solução completa. Da mesma forma, um procedimento que não define o que acontece para uma entrada inválida transfere uma decisão importante para um lugar imprevisível. Especificar limites e casos especiais é parte do raciocínio, não um acabamento posterior.

Tabela 1 — Representações úteis em cada momento

Representação	Pergunta que ajuda a responder	Quando usar

Representação	Pergunta que ajuda a responder	Quando usar
Texto estruturado	Quais são os passos principais?	Planejamento inicial
Pseudocódigo	Como o fluxo de controle funciona?	Antes da implementação
Fluxograma	Quais caminhos podem ser percorridos?	Discussão de decisões e ciclos
Tabela de rastreamento	Como os valores mudam?	Depuração e validação
Código Python	A solução pode ser executada?	Implementação e testes

Exemplo de pseudocódigo. Leia o código com uma entrada pequena e tente antecipar a saída.

```
leia preço
se preço < 0:
   informe "valor inválido"
senão:
   desconto = preço * 0.10
   total = preço - desconto
   informe total
```

Saída esperada

```
A estrutura antecipa uma decisão sem depender da sintaxe de Python.
```

Síntese do capítulo

- Pseudocódigo e fluxogramas tornam o fluxo discutível antes da implementação.
- Casos inválidos e condições de término pertencem ao algoritmo.
- A melhor representação é aquela que torna a dúvida atual mais fácil de responder.

Dados, estado e rastreamento

Durante a execução, um programa mantém um estado: os valores associados às variáveis em determinado instante. Rastrear significa observar como esse estado muda. Para quem está começando, essa prática esclarece erros que não estão no comando isolado, mas na ordem em que os comandos alteram os dados.

Considere um acumulador. Ele começa com um valor inicial, recebe uma contribuição a cada repetição e, ao final, representa um total. Se o valor inicial estiver errado, todas as etapas seguintes podem parecer coerentes e ainda assim produzir uma resposta incorreta. Registrar o estado antes e depois de cada alteração revela a origem do desvio.

Uma tabela de rastreamento deve conter apenas as variáveis relevantes. Colunas demais escondem a história; colunas de menos impedem a conferência. Para um algoritmo que calcula a soma de números, por exemplo, entrada, contador e acumulador normalmente bastam. A tabela serve para enxergar a execução e conferir uma regra.

A previsão manual é uma forma de teste. Antes de rodar um programa, escolha uma entrada pequena e calcule o resultado à mão. Se o programa discordar, compare os passos. Quando a entrada é pequena, é mais fácil distinguir um erro de fórmula, uma condição invertida, um contador que começa no lugar errado ou um valor que não foi atualizado.

Acumulador e contador. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
soma = 0
contador = 1
```

```
while contador <= 4:
    soma = soma + contador
    contador = contador + 1

print(soma)
```

Saída esperada

A tabela mental da execução termina com soma igual a 10 e contador igual a 5.

Síntese do capítulo

- O estado é a fotografia dos valores em um momento da execução.
- Tabelas de rastreamento ajudam a encontrar o primeiro passo incorreto.
- Entradas pequenas são excelentes para testes manuais.

Primeiro conjunto de exercícios

Os exercícios deste capítulo treinam a passagem do enunciado para uma descrição operacional. Não comece pela linguagem. Escreva primeiro as entradas, o resultado esperado e os passos em ordem. Só depois escolha se a solução será expressa em pseudocódigo ou em Python.

Em todos os exercícios, considere também os casos de fronteira. Se o enunciado fala em uma idade, pergunte o que acontece com zero. Se fala em uma quantidade, pergunte o que acontece com uma lista vazia. A intenção não é complicar um exercício simples, mas desenvolver o hábito de explicitar decisões.

Atividade guiada

Para cada proposta, entregue a decomposição do problema, um exemplo de entrada e saída e um algoritmo em pseudocódigo.

1. Calcule o valor total de uma compra com três produtos e uma taxa de entrega.
2. Determine se uma temperatura está abaixo, dentro ou acima de uma faixa de conforto.
3. Leia três notas e descreva como decidir entre aprovado, exame e reprovado.
4. Conte quantos minutos existem em um intervalo informado em horas e minutos.
5. Descreva um procedimento para encontrar o maior de quatro valores sem usar funções prontas.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- Todo exercício deve começar com um modelo da entrada e da saída.
- Casos de fronteira fazem parte da especificação.
- Uma solução bem explicada pode ser avaliada antes de virar código.

CAPÍTULO 2

Primeiros passos com Python

A sintaxe é o meio; o raciocínio é o conteúdo.

Interpretador, scripts e indentação

Python pode ser usada de forma interativa ou por meio de um arquivo de script. No modo interativo, cada comando é digitado e avaliado imediatamente. Esse modo é útil para experimentar expressões, consultar tipos e verificar uma hipótese curta. Em um script, os comandos são organizados em um arquivo que pode ser salvo, executado novamente e compartilhado.

A indentação possui significado em Python. Os espaços no início de uma linha delimitam blocos associados a uma função, decisão, repetição ou tratamento de exceção. Por isso, não se deve tratar a indentação apenas como estética. Um bloco recuado pertence à instrução que o introduziu; uma linha alinhada fora do bloco pode mudar a lógica do programa.

Mensagens de erro são parte do ambiente de aprendizagem. Um erro de sintaxe informa que o interpretador não conseguiu formar a estrutura do programa. Um erro em tempo de execução surge quando a estrutura é válida, mas uma operação não pode ser realizada. Um resultado errado sem mensagem é um erro lógico e exige comparação com um resultado esperado.

A documentação oficial da linguagem descreve regras, tipos e bibliotecas. É uma fonte de consulta, não um roteiro que precisa ser memorizado. O hábito profissional é formular uma pergunta específica, procurar o trecho correspondente, testar um exemplo mínimo e registrar a decisão tomada.

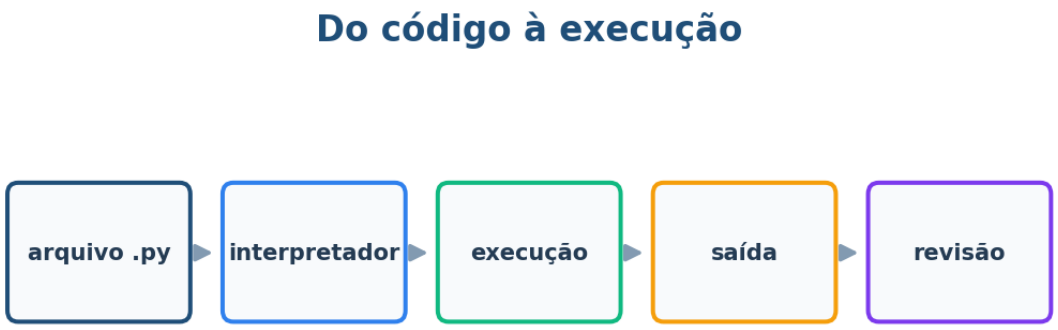


Figura 2 — O caminho de um arquivo Python até a saída

Primeiro script. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
mensagem = "Olá, Computação!"
print(mensagem)
```

Saída esperada

```
Olá, Computação!
```

Síntese do capítulo

- O modo interativo favorece experimentos curtos; o script favorece organização e repetição.
- Indentação define blocos em Python.
- Erros de sintaxe, execução e lógica pedem estratégias de investigação diferentes.

Nomes, comentários e leitura do código

Uma variável é um nome associado a um valor durante a execução. O nome deve comunicar o papel do valor, não apenas o tipo. idade_usuario explica mais do que x quando o programa trabalha com idade. Bons nomes reduzem a necessidade de comentários e tornam a leitura uma forma de verificação.

Comentários devem explicar uma decisão, uma hipótese ou uma restrição que não é evidente no código. Repetir literalmente o que o comando já diz acrescenta ruído. Em um programa didático, comentários também podem chamar a atenção para uma mudança de estado ou para um caso de entrada que merece teste.

A leitura deve acompanhar o fluxo. Comece identificando onde os dados entram, localize as transformações e observe onde a saída é produzida. Em seguida, pergunte quais valores podem chegar a cada ponto. Essa abordagem é mais segura do que tentar interpretar cada linha isoladamente.

O estilo PEP 8 recomenda convenções para tornar programas Python mais consistentes. Convenções não substituem o raciocínio, mas reduzem atritos: nomes previsíveis, linhas com tamanho razoável, espaços bem colocados e funções de responsabilidade limitada.

Tabela 2 — Perguntas para ler um programa

Pergunta	Indício no código	Por que importa
De onde vêm os dados?	input, argumentos ou arquivo	define o espaço de entradas
Onde o estado muda?	atribuições e chamadas	localiza efeitos
Quais caminhos existem?	if, elif, else	revela casos possíveis
Quando termina?	return, condição ou fim do arquivo	evita ciclos ou saídas incompletas
Como verifico?	print, assert ou teste	compara comportamento e expectativa

Síntese do capítulo

- Nomes expressivos funcionam como uma documentação compacta.
- Comentários devem registrar intenção ou decisão, não narrar cada símbolo.
- Ler código é seguir dados, decisões e mudanças de estado.

Entrada, saída e conversão

A função input lê texto. Mesmo quando o usuário digita um número, o valor retornado é uma cadeia de caracteres. A conversão precisa ser explícita porque o programa deve decidir se aceita inteiros, números reais ou apenas textos. Separar leitura e conversão deixa o ponto de validação visível.

A saída também faz parte do contrato do programa. Uma mensagem precisa informar o que está sendo apresentado e usar unidades quando necessário. Resultado: 10 é menos claro do que Tempo total: 10 minutos. Em programas maiores, uma saída clara reduz erros de interpretação e facilita testes automatizados.

F-strings permitem combinar texto e valores de forma legível. Ainda assim, a formatação não deve esconder cálculos longos. Calcule em variáveis com nomes compreensíveis e use a f-string apenas para apresentar o resultado. Isso torna possível testar o cálculo sem depender da aparência da mensagem.

Entradas inválidas serão tratadas em capítulo posterior. Por enquanto, o leitor deve reconhecer que conversões podem falhar e que um programa didático precisa declarar sua expectativa: qual formato será aceito, qual unidade será usada e como o usuário saberá que digitou algo incorreto.

Leitura, conversão e apresentação. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
nome = input("Nome: ")
idade = int(input("Idade: "))
proximo_ano = idade + 1

print(f"{nome} terá {proximo_ano} anos no próximo ano.")
```

Saída esperada

```
Ana terá 21 anos no próximo ano.
```

Síntese do capítulo

- input retorna texto; a conversão define o tipo que o programa espera.
- Uma saída útil informa contexto e unidade.
- Separar cálculo de apresentação torna o programa mais fácil de testar.

Exercícios de entrada e saída

Antes de escrever o código, faça uma previsão para um caso comum e outro de fronteira. Depois de implementar, altere os valores para confirmar que o programa não depende de um exemplo específico.

Evite adicionar menus, classes e bibliotecas a exercícios que pedem uma transformação simples. A solução mais curta que expressa a regra com clareza é um bom ponto de partida. Complexidade deve ser introduzida quando resolve uma necessidade real.

Prática de implementação

Escreva scripts pequenos e verifique cada resultado com uma conta independente.

6. Converta quilômetros em metros e centímetros.
7. Calcule o preço final de um produto com desconto informado pelo usuário.
8. Leia base e altura de um triângulo e apresente a área com duas casas decimais.
9. Leia uma quantidade de segundos e converta para horas, minutos e segundos.
10. Leia o nome de uma disciplina e três notas; apresente a média formatada.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Exemplos de execução antecipam o contrato do script.
- Soluções pequenas favorecem a observação da lógica.
- Testar com valores diferentes evita respostas acidentalmente corretas.

CAPÍTULO 3

Dados e expressões

Uma expressão é uma pergunta calculável sobre valores.

Tipos básicos e significado

O tipo de um valor informa quais operações fazem sentido. Inteiros representam contagens ou posições; números reais representam medidas que podem possuir parte fracionária; valores booleanos representam condições; strings representam texto. Escolher o tipo adequado é uma decisão de modelagem, não apenas uma exigência da linguagem.

A mesma aparência pode esconder significados diferentes. A string 2026 é um texto, enquanto o inteiro 2026 pode participar de uma soma. Um código postal pode parecer numérico, mas normalmente deve ser tratado como texto porque zeros à esquerda são relevantes e operações aritméticas não fazem sentido.

Python é dinamicamente tipada: o nome não precisa ser declarado com um tipo fixo. Isso torna a escrita inicial rápida, mas não elimina a responsabilidade de manter significado estável. Reutilizar o mesmo nome para texto, número e lista em poucas linhas torna o rastreamento difícil e aumenta a chance de erro.

Quando um programa recebe dados de uma fonte externa, o tipo real precisa ser conferido. A função type ajuda em experimentos, mas a decisão principal deve vir do domínio do problema. Pergunte sempre: este valor será somado, comparado, percorrido, formatado ou usado como identificador?

Tabela 3 — Tipos comuns em programas introdutórios

Tipo	Exemplo	Uso típico
int	42	contagens, índices, quantidades inteiras
float	3.14	medidas e cálculos com fração
bool	True	condições e estados binários
str	Python	nomes, mensagens e códigos
list	[2, 4, 6]	sequências mutáveis; será aprofundada depois

Síntese do capítulo

- Tipo é parte do significado de um dado.
- Nem todo valor com aparência numérica deve ser usado em cálculos.
- A flexibilidade de Python exige disciplina na escolha e no uso de nomes.

Operadores, precedência e conversões

Operadores combinam valores para produzir novos valores. Operadores aritméticos expressam cálculos; operadores relacionais produzem condições; operadores lógicos combinam condições. Ler uma expressão exige identificar a ordem de avaliação e o tipo do resultado.

A precedência segue regras da linguagem, mas parênteses são recomendáveis quando tornam a intenção mais clara. Uma expressão curta pode ser correta e ainda assim difícil de revisar se depender de uma regra que o leitor precisa lembrar. O código deve comunicar a fórmula, não funcionar como um teste de memória.

Conversões como int, float e str alteram a representação de um valor. Elas não corrigem automaticamente um dado inadequado. Converter 12.5 diretamente para int falha porque o texto não representa um inteiro. Converter 12.5 para inteiro descarta a parte fracionária, o que pode ou não ser aceitável no problema.

Uma operação deve ser analisada pelos seus limites. Dividir por zero produz erro; transformar texto não numérico em número produz erro; misturar tipos incompatíveis pode produzir uma mensagem ou uma concatenação inesperada. Conhecer os limites permite escrever validações melhores.

Precedência explícita. Faça uma previsão, execute e compare os dois resultados.

```
total = 7 + 3 * 2
total_com_parenteses = (7 + 3) * 2

print(total)
print(total_com_parenteses)
```

Saída esperada

```
13
20
```

Síntese do capítulo

- A precedência define a ordem de avaliação; parênteses comunicam intenção.
- Conversão é uma decisão semântica, não um reparo universal.
- Limites das operações devem aparecer no planejamento e nos testes.

Expressões booleanas

Uma expressão booleana responde a uma pergunta que pode ser verdadeira ou falsa. Ela pode comparar valores, testar pertencimento ou combinar outras condições. O resultado booleano é a matéria-prima das estruturas de decisão e repetição.

As palavras and, or e not devem ser lidas como relações entre condições. Acesso permitido se usuário ativo e senha correta é diferente de ativo ou senha correta. Escrever as frases antes da expressão ajuda a evitar a inversão de uma regra.

A avaliação de curto-circuito significa que Python pode deixar de avaliar parte de uma expressão quando o resultado já é conhecido. Em uma conjunção, uma condição falsa torna o resultado falso; em uma disjunção, uma condição verdadeira torna o resultado verdadeiro. Esse comportamento pode proteger operações que só devem ocorrer depois de uma verificação.

Comparações encadeadas podem ser legíveis para intervalos. Ainda assim, o leitor deve confirmar se as extremidades são inclusivas. A diferença entre idade >= 18 e idade > 18 muda o tratamento de uma pessoa de exatamente 18 anos. Fronteiras merecem exemplos explícitos.

Condição composta. Observe como a entrada passa por cada etapa até chegar à saída.

```
idade = 20
documento = True
pode_entrar = idade >= 18 and documento

print(pode_entrar)
```

Saída esperada

```
True
```

Síntese do capítulo

- Booleanos representam respostas a perguntas do domínio.
- A ordem e o operador usado devem refletir a frase da regra.
- Fronteiras como igualdade e desigualdade devem ser testadas.

Exercícios com expressões

A tarefa deste conjunto é praticar a tradução de regras para expressões. Escreva a regra em português, a expressão em Python e pelo menos três testes: um verdadeiro, um falso e um caso de fronteira.

Oficina de tipos e condições

Formule e teste as expressões sem colocar toda a lógica em uma única linha extensa.

11. Verifique se uma nota está entre 0 e 10, inclusive.
12. Verifique se um ano é bissexto usando a regra do calendário gregoriano.
13. Verifique se uma pessoa pode solicitar um serviço quando possui idade mínima e cadastro ativo.
14. Verifique se três medidas podem representar um triângulo.
15. Verifique se um texto está vazio ou contém apenas espaços.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- Toda condição deve ser acompanhada de exemplos.
- Caso de fronteira é parte da especificação.
- Expressões claras são preferíveis a expressões apenas compactas.

CAPÍTULO 4

Decisões e caminhos

Decidir é escolher um caminho com base no estado atual.

A estrutura if

A estrutura if executa um bloco somente quando uma condição é verdadeira. Ela liga uma expressão booleana a uma consequência. Essa relação deve ser lida como uma frase: se a condição acontecer, faça isto. O bloco precisa conter apenas ações que dependem daquela condição.

A alternativa else representa o caminho restante, mas não necessariamente todos os casos possíveis do domínio. Se a condição é nota maior ou igual a 7, o else inclui qualquer valor abaixo de 7, inclusive uma entrada inválida, caso a validação não tenha acontecido antes. A estrutura de decisão deve ser organizada na ordem que separa validade e classificação.

O elif permite testar alternativas em sequência. A ordem importa porque somente o primeiro bloco cuja condição seja verdadeira será executado. Se uma condição ampla vier antes de uma condição específica, o caso específico pode nunca ser alcançado.

Uma decisão bem escrita torna os caminhos visíveis. Evite expressões que misturam conversão, cálculo e classificação em uma única condição. Nomeie condições importantes e use blocos pequenos. Isso facilita a leitura, a revisão e os testes.

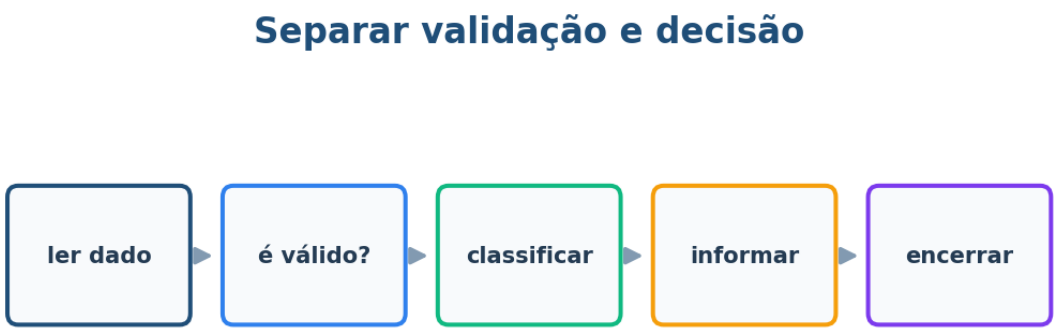


Figura 3 — Uma decisão com caminhos alternativos

Validação antes da classificação. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

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

if nota < 0 or nota > 10:
   print("Nota inválida")
elif nota >= 7:
   print("Aprovado")
else:
   print("Reprovado")
```

Saída esperada

```
Nota inválida, Aprovado ou Reprovado, conforme a entrada.
```

Síntese do capítulo

- Validade do dado e classificação do resultado são decisões diferentes.
- A ordem das condições altera o caminho percorrido.
- Nomear condições e manter blocos pequenos melhora a revisão.

Decisões aninhadas e guard clauses

Decisões aninhadas aparecem quando uma condição só faz sentido depois de outra. Por exemplo, não se deve calcular o percentual de desconto de um cadastro inexistente. O primeiro teste protege o acesso ao dado; o segundo aplica a regra comercial.

Muitos níveis de aninhamento tornam o fluxo difícil de acompanhar. Uma alternativa é usar guard clauses: verificar uma situação inválida no início e retornar imediatamente, deixando o caminho principal menos recuado. Em programas introdutórios, isso pode ser feito com return dentro de funções ou com uma sequência organizada de validações.

A escolha entre uma sequência de if, um if com condições combinadas e uma tabela de regras depende da quantidade de casos. Duas condições simples podem ser combinadas; muitas faixas numéricas pedem uma ordem explícita; regras que mudam com frequência podem ser representadas por dados.

Não existe mérito em usar uma estrutura sofisticada para um problema curto. O critério é a facilidade de responder: para cada entrada relevante, é possível dizer qual bloco executa? Se a resposta exige simular muitos níveis, simplifique a representação.

Tabela 4 — Estratégias para organizar decisões

Estratégia	Adequada quando	Risco a observar
if / else	há dois caminhos	tratar inválido como alternativa válida
if / elif / else	há classes mutuamente exclusivas	ordem de condições
condições guardas	há entradas que interrompem o fluxo	mensagens incompletas
regras em dados	há muitas faixas ou parâmetros	dados sem validação

Síntese do capítulo

- Aninhamento deve refletir dependência entre decisões.
- Guard clauses reduzem caminhos inválidos no corpo principal.
- A estrutura deve ser escolhida pelo número e pela natureza dos casos.

Testar decisões

Uma decisão precisa de testes que cubram seus caminhos. Se um programa possui três saídas, um único exemplo não demonstra correção. O conjunto mínimo deve conter uma entrada para cada caminho e, quando houver limites, entradas imediatamente abaixo, exatamente sobre e imediatamente acima do limite.

A tabela de decisão é uma ferramenta simples para organizar cobertura. Liste condições relevantes, indique combinações plausíveis e registre a saída esperada. Ela não substitui testes automatizados, mas revela lacunas antes que o código seja escrito.

Testes também devem verificar mensagens e unidades quando elas fazem parte do contrato. Um programa pode calcular o valor correto e informar uma unidade errada. Em um material didático, esse cuidado ensina que correção inclui resultado, formato e comunicação.

Quando um teste falha, procure o primeiro ponto em que a execução se afasta da expectativa. Corrigir a linha final que exibiu o valor errado pode mascarar a causa. O diagnóstico deve acompanhar dados e decisões desde a entrada.

Casos de fronteira. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
def classificar_nota(nota):
    if nota < 0 or nota > 10:
        return "inválida"
    if nota >= 7:
        return "aprovado"
    if nota >= 5:
        return "exame"
    return "reprovado"

for valor in (-1, 0, 4.9, 5, 6.99, 7, 10, 11):
    print(valor, classificar_nota(valor))
```

Saída esperada

```
A saída percorre todas as classes e testa os limites.
```

Síntese do capítulo

- Cada caminho observável precisa de ao menos um teste.
- Limites devem ser testados abaixo, sobre e acima.
- A falha deve ser investigada desde a primeira divergência.

Exercícios de decisão

Nas propostas seguintes, escreva primeiro uma tabela de casos. Depois implemente uma função que receba os dados e retorne uma classificação, deixando a apresentação para o programa principal.

Prática de decisões

Implemente as regras com funções curtas e teste todos os caminhos.

16. Classifique um número como positivo, negativo ou zero.
17. Calcule uma tarifa com faixas de consumo e trate consumo negativo.
18. Determine o tipo de um triângulo a partir dos lados válidos.
19. Decida se uma data é válida para um mês informado.
20. Crie uma calculadora que valide o operador e impeça divisão por zero.

Para conferir. Altere os valores do exemplo e confirme se a regra continua válida.

Síntese do capítulo

- Tabelas de casos ajudam a transformar regras em testes.
- Funções que retornam classificação são mais fáceis de verificar.
- Entradas inválidas devem ser tratadas intencionalmente.

CAPÍTULO 5

Repetições e padrões

Repetir é automatizar um passo que mantém a mesma regra.

for e range

O laço for percorre uma sequência de valores. Em exercícios introdutórios, range fornece uma sequência de inteiros útil para repetir uma operação um número conhecido de vezes. A pergunta central é: quais valores serão percorridos? Antes de executar, escreva o primeiro e o último valor e confirme que o limite final de range é exclusivo.

O nome usado no laço representa o valor da iteração atual. Ele não é necessariamente um índice, embora possa ser. Se o programa precisa apenas repetir uma ação, um nome como repeticao pode ser mais honesto do que i. Se precisa acessar posições, o índice deve ser justificado.

Percorrer uma sequência e repetir uma quantidade fixa são ideias relacionadas, mas diferentes. O primeiro depende de valores existentes; o segundo depende de uma contagem. Confundir os dois pode levar a acessos fora do limite ou a itens ignorados.

O corpo do laço deve expressar uma ação que se repete. Cálculos que só precisam ocorrer uma vez devem ficar fora dele. Essa separação evita trabalho desnecessário e ajuda a identificar o estado que realmente muda em cada iteração.

Percorrendo uma faixa. Depois de entender a versão básica, experimente uma entrada diferente.

```
for numero in range(1, 6):
    quadrado = numero ** 2
    print(numero, quadrado)
```

Saída esperada

```
São produzidos os quadrados de 1 a 5.
```

Síntese do capítulo

- range possui limite final exclusivo.
- Diferencie repetição por contagem de percurso de uma sequência.
- O corpo deve conter somente o trabalho que se repete.

while e repetição controlada por condição

O laço while é apropriado quando a quantidade de repetições depende de uma condição que pode mudar durante a execução. O programa precisa inicializar o estado, testar a condição, executar o corpo e atualizar o estado de modo que a condição eventualmente se torne falsa.

Uma repetição controlada por sentinela continua até que um valor especial seja recebido. A sentinela não é um dado comum do processamento; ela é um comando embutido na entrada. Por isso, deve ser documentada e não deve ser contabilizada como parte dos dados.

O principal risco é o ciclo infinito. Ele ocorre quando a condição nunca se torna falsa ou quando a variável que deveria mudá-la não é atualizada. Durante o desenvolvimento, use entradas curtas e registre o estado de cada repetição.

Quando o programa precisa executar pelo menos uma leitura antes de decidir se continua, o padrão de leitura inicial seguida de leitura ao final do corpo pode ser adequado. Não há problema em repetir uma instrução quando isso torna a condição explícita; há problema quando a duplicação esconde uma regra.

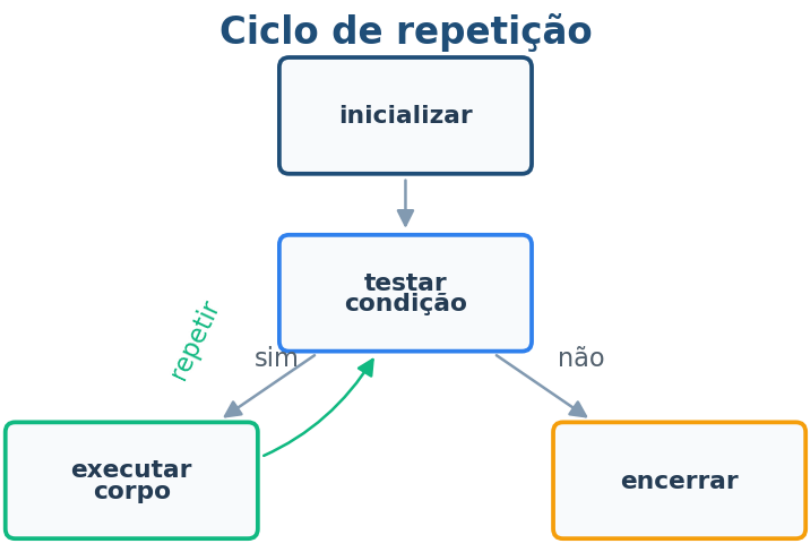


Figura 4 — Etapas de um ciclo while

Repetição com sentinela. Preste atenção ao ponto em que o programa decide, repete ou transforma os dados.

```
total = 0
valor = float(input("Valor (0 encerra): "))

while valor != 0:
   total += valor
   valor = float(input("Valor (0 encerra): "))

print(f"Total: {total:.2f}")
```

Saída esperada

```
O zero encerra a leitura e não entra no total.
```

Síntese do capítulo

- while exige uma condição e uma atualização de estado.
- A sentinela controla o fluxo e não deve ser tratada como dado comum.
- Entradas pequenas ajudam a detectar ciclos que não terminam.

Acumuladores, contadores e extremos

Acumulador é uma variável que reúne contribuições ao longo de um percurso. Contador registra quantas vezes uma condição ocorreu. Embora os dois padrões usem atualização dentro de laços, seus significados são diferentes: um soma valores, o outro soma ocorrências.

Para calcular média, acumule a soma e conte os valores. A divisão deve acontecer depois da repetição, pois somente então o total de dados é conhecido. Se nenhuma entrada válida for encontrada, o programa precisa definir o que fazer antes de dividir.

Encontrar mínimo e máximo exige cuidado com a inicialização. Usar zero como melhor valor inicial falha quando todos os dados são negativos. Uma estratégia geral é usar o primeiro dado válido como referência e comparar os dados seguintes.

Esses padrões podem ser combinados. Um programa pode contar aprovados, acumular notas e guardar a maior nota no mesmo percurso. Entretanto, muitas responsabilidades em um único laço aumentam a carga de leitura. Quando a clareza exigir, divida o processamento em funções ou percursos separados.

Tabela 5 — Padrões de estado em repetições

Padrão	Inicialização	Atualização	Resultado
Soma	0	acumulador += valor	total
Contagem	0	contador += 1	quantidade
Média	soma=0; qtd=0	atualiza soma e quantidade	soma / qtd
Máximo	primeiro valor	se valor > maior	maior
Presença	False	se encontrar: True	existe ou não

Síntese do capítulo

- O nome da variável deve indicar o padrão de estado que ela representa.
- Médias exigem tratar a ausência de dados.
- Extremos devem ser inicializados com uma referência válida.

Laços aninhados e rastreamento

Um laço aninhado é uma repetição dentro de outra repetição. Ele aparece quando cada elemento de um conjunto precisa ser comparado ou combinado com elementos de outro conjunto, ou quando o problema possui linhas e colunas. O laço interno completa todas as suas iterações para cada iteração do laço externo.

A tabela de rastreamento deve diferenciar as variáveis dos dois níveis. Registre a variável externa, a interna e o resultado parcial. Esse registro torna visível a ordem de execução e explica por que uma ação pode ocorrer muitas vezes.

Laços aninhados não são necessariamente errados. Eles podem ser a representação direta de uma regra. O cuidado é avaliar o tamanho da entrada: repetir uma operação em duas dimensões pode ser suficiente para dezenas de elementos e inadequado para milhões.

Quando um padrão aparece várias vezes, pergunte se existe uma forma de reduzir o trabalho ou usar uma estrutura de dados apropriada. Essa pergunta será retomada no segundo livro, mas já pode orientar a análise de programas.

Duas dimensões. Rode o exemplo uma vez e descreva, com suas palavras, o que aconteceu.

```
for linha in range(1, 4):
    for coluna in range(1, 4):
        produto = linha * coluna
```

```
        print(f"{linha} x {coluna} = {produto}")
```

Saída esperada

```
Para cada linha, todas as colunas são processadas.
```

Síntese do capítulo

- O laço interno reinicia a cada nova iteração do laço externo.
- Rastreamento com duas variáveis revela a ordem dos eventos.
- A necessidade de um laço aninhado deve ser avaliada junto ao tamanho da entrada.

Exercícios de repetição

Em cada problema, defina com clareza o que inicia o ciclo, o que mantém o ciclo e o que faz o ciclo terminar. Escolha for quando a sequência ou quantidade for conhecida; escolha while quando a continuação depender de uma condição atualizada.

Laboratório de laços

Implemente, teste e explique o papel de cada variável de estado.

21. Some os números pares entre dois limites.
22. Conte vogais em uma palavra sem usar uma função pronta de contagem.
23. Leia notas até a sentinela -1 e apresente média e quantidade.
24. Verifique se um número é primo usando divisores possíveis.
25. Imprima uma tabela de multiplicação para um intervalo escolhido.

Como conferir. Releia o enunciado e verifique se cada condição foi contemplada.

Síntese do capítulo

- Toda repetição precisa de uma condição de continuidade e de término.
- O padrão de estado deve ser identificável pelo nome das variáveis.
- O custo de laços aninhados depende do tamanho da entrada.

CAPÍTULO 6

Funções e modularização

Uma função é uma pequena promessa: recebe algo, faz uma tarefa e devolve um resultado.

Abstração e responsabilidade

Uma função agrupa instruções sob um nome. O nome permite substituir uma sequência de detalhes por uma intenção: calcular_media comunica o que importa para quem chama, enquanto os passos internos ficam localizados. Essa é a essência da abstração.

Uma função deve ter uma responsabilidade compreensível. Isso não significa que ela tenha sempre poucas linhas, mas que suas linhas respondam à mesma pergunta. Uma função que lê, calcula, grava, imprime e decide pode funcionar, porém será difícil reutilizar e testar.

Parâmetros tornam a função adaptável. O argumento representa uma entrada que varia; o corpo aplica uma regra; o retorno disponibiliza o resultado. O contrato da função deve dizer o que cada parâmetro significa e quais valores são aceitos.

Funções não precisam ser criadas apenas quando existe repetição textual. Elas também servem para nomear uma regra complexa, limitar o alcance de uma variável e criar um ponto de teste. Modularizar é organizar raciocínio, não somente evitar copiar e colar.

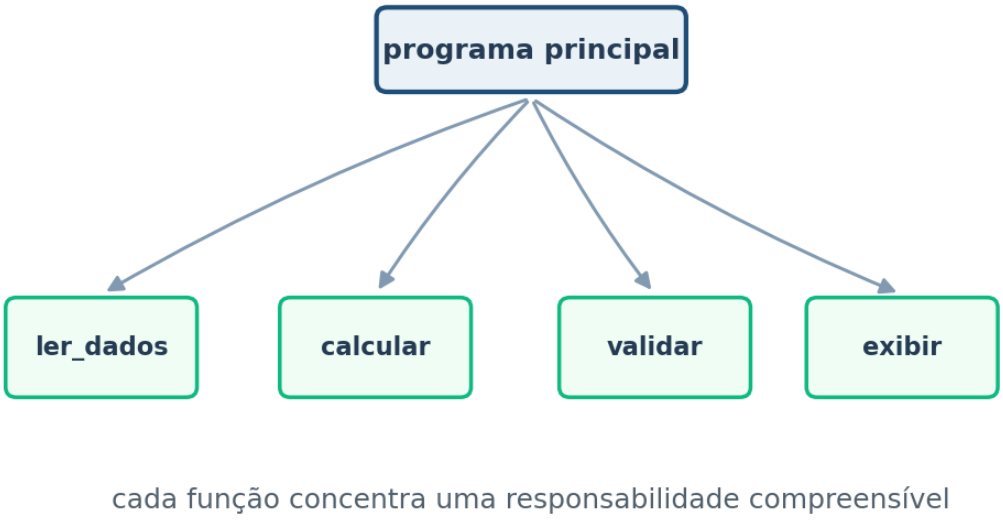


Figura 5 — Decomposição de um programa em funções

Função com parâmetros e retorno. Compare a saída com o contrato descrito no texto anterior.

```
def calcular_media(nota1, nota2, nota3):
    return (nota1 + nota2 + nota3) / 3

media = calcular_media(7, 8, 9)
print(f"Média: {media:.1f}")
```

Saída esperada

```
Média: 8.0
```

Síntese do capítulo

- Abstração permite falar em intenção sem repetir detalhes.
- Responsabilidade clara favorece teste e reutilização.
- Parâmetros e retorno formam o contrato de uma função.

Escopo e efeitos

Escopo é a região do programa em que um nome pode ser utilizado. Variáveis criadas dentro de uma função normalmente pertencem àquela função. Essa separação evita que uma alteração local modifique silenciosamente todo o programa.

Uma função pura depende apenas de seus argumentos e produz um retorno sem alterar estado externo. Funções com efeitos colaterais leem entrada, imprimem mensagens, alteram arquivos ou modificam objetos compartilhados. Efeitos são necessários em programas reais, mas devem aparecer em pontos bem definidos.

Separar cálculo de entrada e saída é uma técnica didática e profissional. A função pode receber valores já lidos e retornar uma resposta; outra parte do programa conversa com o usuário. Assim, o cálculo pode ser testado com chamadas diretas, sem simular teclado.

Quando um efeito é inevitável, documente-o pelo nome e pela posição do programa. Uma função chamada salvar_relatorio pode gravar um arquivo; uma função chamada calcular_total não deveria fazê-lo sem que isso seja evidente. Nomes e responsabilidades trabalham juntos.

Tabela 6 — Função mais fácil de testar

Característica	Função concentrada em cálculo	Função concentrada em interação
Entrada	parâmetros	input, arquivo ou serviço
Saída	return	print, arquivo ou atualização externa
Teste	chamada direta	preparação de ambiente
Reutilização	alta	depende do contexto

Síntese do capítulo

- Escopo limita onde nomes podem ser usados.
- Cálculos puros são pontos de teste econômicos.
- Efeitos externos devem ser visíveis pela função e pela organização do programa.

Decomposição passo a passo

Para decompor uma tarefa, comece pela saída desejada e pergunte quais informações precisam ser produzidas. Depois identifique cálculos, validações e interações. Cada resposta pode se transformar em uma função, desde que possua uma responsabilidade coerente.

Uma estratégia útil é escrever uma versão linear do algoritmo, executá-la com um exemplo e marcar trechos que respondem a perguntas diferentes. Em seguida, extraia uma função por vez. Após cada extração, execute os mesmos testes. A modularização deve preservar o comportamento enquanto melhora a organização.

Funções pequenas não são um objetivo matemático. Se separar uma operação exige passar muitos argumentos e torna o fluxo artificial, a fronteira escolhida pode estar errada. A divisão deve reduzir a complexidade percebida pelo leitor.

O programa principal deve parecer uma narrativa curta: ler dados, validar, calcular, apresentar. Quando o fluxo principal está cheio de detalhes internos, ele deixa de explicar o que o programa faz. Quando tudo foi escondido em funções genéricas, o leitor perde a história. O equilíbrio é uma habilidade construída pela revisão.

Fluxo principal legível. Leia o código com uma entrada pequena e tente antecipar a saída.

```
def ler_temperatura():
    return float(input("Temperatura em °C: "))

def converter_para_fahrenheit(celsius):
    return celsius * 9 / 5 + 32

temperatura = ler_temperatura()
resultado = converter_para_fahrenheit(temperatura)
print(f"{resultado:.1f} °F")
```

Saída esperada

```
A leitura, a transformação e a saída possuem responsabilidades separadas.
```

Síntese do capítulo

- Extraia funções preservando o comportamento observado.
- O programa principal deve comunicar a narrativa da solução.
- Modularização exige revisão do contrato, não apenas recorte de linhas.

Exercícios de funções

Escreva funções com nomes verbais, parâmetros explícitos e retorno quando houver um valor que possa ser testado. Não coloque input dentro das funções de cálculo. Teste cada função com valores normais e de fronteira.

Oficina de modularização

Divida cada problema em funções e acrescente uma pequena bateria de chamadas de teste.

26. Crie funções para converter temperatura entre Celsius, Fahrenheit e Kelvin.
27. Crie uma função que calcule o valor de uma prestação com juros simples.
28. Crie funções para validar e formatar um horário.
29. Crie uma função que retorne a maior e a menor de três notas.
30. Crie um programa de menu em que cada opção chama uma função específica.

Verificação. Acompanhe o contador ou acumulador em uma tabela curta.

Síntese do capítulo

- Funções são fronteiras de entendimento e de teste.
- Parâmetros devem representar dados necessários à regra.
- A interação com usuário não precisa contaminar o cálculo.

CAPÍTULO 7

Textos e coleções simples

Representar bem os dados reduz o esforço do algoritmo.

Strings como sequências de caracteres

Uma string representa texto, mas também pode ser percorrida como uma sequência de caracteres. Índices permitem acessar posições e fatias permitem extrair intervalos. O primeiro índice é zero, uma convenção que aparece em muitas linguagens e precisa ser observada desde o início.

Strings são imutáveis em Python: uma operação que parece alterar o texto produz uma nova string. Essa característica torna o comportamento previsível, mas exige atribuir o resultado quando uma transformação deve ser preservada.

Métodos como strip, lower, split e join resolvem operações comuns. O leitor deve conhecer a diferença entre uma função que cria uma nova string e uma operação que modifica uma coleção mutável. Também deve verificar como espaços, maiúsculas e caracteres especiais afetam a comparação.

Textos de entrada quase sempre precisam de normalização. Remover espaços das extremidades, padronizar caixa e validar comprimento são decisões do domínio. Não normalize indiscriminadamente: em uma senha, por exemplo, alterar maiúsculas ou espaços pode mudar o valor correto.

Normalização de texto. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
frase = "  Python ajuda a pensar.  "
limpa = frase.strip()
palavras = limpa.lower().split()

print(limpa)
print(len(palavras))
```

Saída esperada

```
O texto é limpo, convertido para minúsculas e separado em palavras.
```

Síntese do capítulo

- Índices começam em zero.
- Strings são imutáveis; transformações retornam novos valores.
- Normalização deve refletir a regra do problema.

Listas, tuplas, conjuntos e dicionários

Uma lista representa uma sequência mutável e ordenada. Ela é adequada quando a posição e a ordem dos itens importam e quando o programa precisa acrescentar, remover ou substituir elementos. Neste livro, a lista será apresentada apenas como uma ferramenta básica; o segundo livro estudará sua implementação e suas relações com outras estruturas.

Tuplas representam agrupamentos ordenados que não deveriam ser alterados durante o uso. Conjuntos representam itens distintos e são úteis para testes de pertencimento. Dicionários associam chaves a valores e permitem modelar registros simples, como o cadastro de uma pessoa ou a configuração de um serviço.

A escolha da coleção deve partir das operações necessárias. Se a pergunta principal é este item aparece?, um conjunto pode comunicar melhor a intenção. Se é qual valor corresponde a esta chave?, um dicionário é natural. Se é qual é o terceiro item?, uma sequência indexada é apropriada.

Coleções aninhadas permitem representar dados mais ricos, mas também aumentam a necessidade de nomes e invariantes. Antes de criar uma lista de dicionários dentro de outra lista, desenhe um exemplo pequeno e descreva o significado de cada nível.

Tabela 7 — Qual coleção comunica melhor a intenção?

Coleção	Propriedade central	Pergunta típica
list	ordem e mutabilidade	quais itens estão na sequência?
tuple	agrupamento ordenado	quais valores formam este registro fixo?
set	unicidade e pertencimento	este elemento já está presente?
dict	associação chave-valor	qual valor corresponde a esta chave?

Síntese do capítulo

- Estrutura de dados é uma decisão sobre operações e significado.
- Listas, tuplas, conjuntos e dicionários não são intercambiáveis em todos os problemas.
- Um exemplo pequeno ajuda a descobrir se a representação está correta.

Percursos e transformação

Percorrer uma coleção significa visitar seus elementos segundo uma ordem. O laço for esconde detalhes do índice quando a aplicação só precisa do valor. Quando posição e valor são necessários, enumerate torna essa relação explícita.

Uma transformação cria um resultado a partir dos itens de uma coleção. Ela pode ser feita com um laço e uma lista de saída, uma abordagem que costuma ser mais clara para iniciantes. Compreensões de lista são uma forma compacta, mas devem ser usadas quando continuam legíveis.

Filtrar é selecionar itens que atendem a uma condição. Somar, contar ou buscar também podem ser vistos como percursos com um padrão de estado. Reconhecer essas operações ajuda a planejar funções e a comparar soluções.

Um percurso não deve modificar a coleção de modo imprevisível enquanto a percorre. Para remover itens com segurança, crie uma nova sequência ou percorra uma cópia quando isso fizer sentido. A regra geral é saber qual é a coleção de entrada e qual é a coleção de saída.

Filtrando valores. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
notas = [6.5, 8.0, 9.0, 5.5]
aprovadas = []

for nota in notas:
    if nota >= 7:
        aprovadas.append(nota)

print(aprovadas)
```

Saída esperada

```
[8.0, 9.0]
```

Síntese do capítulo

- enumerate associa posição e valor sem controlar índices manualmente.
- Transformar e filtrar são operações recorrentes sobre sequências.
- Modificar uma coleção durante o percurso exige cuidado explícito.

Exercícios de representação

Escolha a coleção antes de implementar. Explique qual operação é mais frequente e por que a estrutura escolhida a torna clara. A atividade serve para praticar modelagem, não para memorizar métodos.

Laboratório de coleções simples

Implemente as tarefas e descreva a representação adotada.

31. Remova espaços repetidos de uma frase e reconstrua o texto.
32. Leia nomes até uma sentinela e mantenha apenas nomes únicos.
33. Represente três produtos com código, descrição e preço.
34. Conte a frequência de cada palavra de uma frase.
35. Separe uma lista de números em pares e ímpares.

Depois do teste. Teste a entrada mínima, vazia ou ausente quando ela fizer sentido.

Síntese do capítulo

- A coleção escolhida deve refletir a pergunta principal do problema.
- Transformações e filtros podem ser descritos como percursos.
- A representação deve ser explicada junto com o código.

CAPÍTULO 8

Arquivos, exceções e depuração

Um programa confiável aprende a lidar com entradas que não seguem o plano ideal.

Leitura e gravação de texto

Arquivos permitem preservar dados entre execuções. Um arquivo de texto possui linhas e caracteres; cabe ao programa definir o formato que será lido. Mesmo um formato simples precisa documentar separadores, ordem dos campos e tratamento de linhas vazias.

O uso de with open organiza o ciclo de vida do arquivo. O modo de abertura define se o programa lerá, substituirá ou acrescentará conteúdo. A codificação deve ser escolhida de forma consciente quando houver acentos e caracteres fora do conjunto básico.

A leitura pode ser feita linha a linha para reduzir memória e tornar o processamento progressivo. Quando o arquivo é pequeno, read ou readlines podem ser suficientes. O critério é relacionar a estratégia ao tamanho, ao formato e às operações que serão realizadas.

Arquivos externos não devem ser considerados confiáveis. Uma linha pode estar incompleta, conter separador extra ou apresentar um valor que não pode ser convertido. A função de leitura deve validar o que recebeu e comunicar a linha ou o campo problemático.

Gravação de texto. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
linhas = ["Ana,8.0", "Bia,7.5", "Caio,9.0"]

with open("notas.txt", "w", encoding="utf-8") as arquivo:
    for linha in linhas:
        arquivo.write(linha + "\n")
```

Saída esperada

```
O arquivo recebe uma linha para cada registro.
```

Síntese do capítulo

- O formato do arquivo é um contrato.
- with organiza o ciclo de vida do arquivo.
- Dados externos precisam ser validados antes do uso.

Exceções e mensagens de erro

Uma exceção interrompe o fluxo normal porque uma operação não pôde ser concluída. ValueError pode indicar conversão inválida; FileNotFoundError informa que um caminho não foi encontrado; ZeroDivisionError revela divisão por zero. O tipo da exceção orienta o diagnóstico.

O tratamento com try e except deve ser específico. Capturar qualquer exceção e continuar silenciosamente esconde defeitos e pode produzir dados incorretos. Trate a situação que você sabe explicar e, quando necessário, informe ao usuário como corrigir a entrada.

Uma exceção não deve ser usada para substituir uma condição simples que pode ser verificada diretamente. Por outro lado, não é necessário duplicar toda a validação quando a operação já fornece uma exceção significativa. O desenho deve equilibrar clareza, segurança e manutenção.

Mensagens de erro devem indicar o que aconteceu, qual dado causou o problema e qual ação é possível. Uma mensagem técnica pode ser útil para desenvolvimento, mas o usuário final precisa de orientação. Em um livro, mostrar as duas perspectivas ajuda a diferenciar diagnóstico de interface.

Validação com exceção. Faça uma previsão, execute e compare os dois resultados.

```
texto = input("Quantidade: ")

try:
   quantidade = int(texto)
   if quantidade < 0:
      raise ValueError("a quantidade não pode ser negativa")
except ValueError as erro:
   print(f"Entrada inválida: {erro}")
else:
   print(f"Quantidade registrada: {quantidade}")
```

Saída esperada

```
A mensagem orienta o usuário sem esconder a causa.
```

Síntese do capítulo

- O tipo da exceção é uma pista para o diagnóstico.
- Trate exceções específicas e explique a recuperação.
- Uma mensagem útil diferencia o erro técnico da orientação ao usuário.

Depuração como investigação

Depurar não é tentar alterações aleatórias até o resultado mudar. É investigar uma diferença entre o comportamento observado e o comportamento esperado. A primeira ação é reproduzir o problema com uma entrada pequena e estável, para que cada tentativa possa ser comparada.

Depois de reproduzir, observe. Use mensagens temporárias, tabelas de rastreamento ou um depurador para verificar os valores em pontos importantes. A pergunta não é qual linha parece estranha?, mas em que ponto o estado deixou de respeitar a regra?

Formule uma hipótese testável. Se suspeita de uma conversão, altere apenas a conversão ou registre o tipo. Se suspeita de um limite, teste os valores imediatamente próximos. Uma alteração que muda muitas coisas ao mesmo tempo impede saber qual hipótese foi confirmada.

Após corrigir, repita o caso que falhou e execute casos que antes funcionavam. A correção precisa preservar o comportamento válido. Registrar a causa e a solução cria um repertório que reduz o tempo de futuras investigações.

Figura 6 — Ciclo de depuração

Síntese do capítulo

- Depuração começa com uma falha reproduzível.
- Hipóteses devem ser testadas com alterações pequenas.
- Depois de corrigir, repita o caso problemático e os casos válidos.

Testes simples e assert

Um teste é uma chamada com uma expectativa. A instrução assert interrompe a execução quando uma condição não é verdadeira, tornando visível uma quebra do contrato. Ela é apropriada para verificações internas e exemplos didáticos, enquanto sistemas maiores podem usar frameworks de teste.

Testar uma função pequena é mais fácil quando ela não imprime nem lê diretamente. Por isso, a separação entre cálculo e interação reaparece durante a depuração. Uma função que retorna um valor pode ser comparada com vários casos em poucas linhas.

A quantidade de testes não é o único critério. Casos representativos cobrem caminhos, limites e entradas incomuns. Um programa de classificação de notas precisa testar cada faixa; um conversor precisa testar zero, valores negativos permitidos e precisão esperada.

Testes são exemplos executáveis da regra. Quando um teste falha, ele também serve como documentação do comportamento esperado. Manter testes claros é uma forma de preservar decisões durante mudanças futuras.

Teste mínimo. Observe como a entrada passa por cada etapa até chegar à saída.

```
def dobro(valor):
    return valor * 2

assert dobro(0) == 0
assert dobro(4) == 8
assert dobro(-3) == -6
print("testes concluídos")
```

Saída esperada

```
testes concluídos
```

Síntese do capítulo

- assert transforma uma expectativa em uma verificação executável.
- Casos de fronteira e caminhos diferentes importam mais do que repetição.
- Testes também documentam o comportamento esperado.

Exercícios de arquivos e depuração

Faça os exercícios em duas etapas. Primeiro implemente a versão que funciona com dados perfeitos. Depois introduza entradas vazias, campos ausentes, valores inválidos e arquivos inexistentes. O objetivo é perceber como a solução muda quando o mundo real deixa de obedecer ao exemplo ideal.

Prática de robustez

Implemente mensagens de erro claras e registre quais casos foram testados.

36. Leia um arquivo de temperaturas e calcule média, mínimo e máximo.
37. Leia um arquivo de produtos e rejeite linhas sem preço válido.
38. Crie uma função que converta texto em inteiro e retorne uma mensagem de erro controlada.
39. Escreva testes para uma função que classifica uma senha por comprimento e composição.
40. Introduza deliberadamente um erro em um programa curto e documente o ciclo de depuração.

Para conferir. Explique em uma frase por que o resultado atende ao enunciado.

Síntese do capítulo

- Robustez nasce de conhecer as entradas que podem falhar.
- Mensagens e testes fazem parte do comportamento do programa.
- Documentar a investigação transforma erro em aprendizagem.

CAPÍTULO 9

Projetos integradores

Um projeto pequeno reúne conceitos suficientes para tornar o raciocínio visível.

Projeto um Calculadora de orçamento

O primeiro projeto integra entrada, conversão, validação, decisões e funções. O programa recebe itens de uma compra, calcula subtotal, aplica uma regra de desconto e apresenta o valor final. A solução deve começar com poucos itens e evoluir apenas quando o contrato estiver claro.

Separe a função que calcula o total da função que interage com o usuário. A primeira pode receber uma lista de preços e uma taxa; a segunda pode construir essa lista. Mesmo que a coleção seja simples neste momento, a separação prepara o programa para mudanças.

Teste valores que ficam exatamente nos limites do desconto. Também teste lista vazia, preço negativo e taxa inválida. Decida se o programa rejeita a entrada ou se aceita valores específicos. Uma decisão documentada é melhor do que um comportamento acidental.

Ao finalizar, escreva uma pequena explicação: quais entradas existem, qual regra foi implementada, quais funções foram criadas e quais testes demonstram os casos principais.

Núcleo do projeto. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

```
def calcular_total(precos, taxa_entrega, desconto_minimo):
    subtotal = sum(precos)
    if subtotal >= desconto_minimo:
        subtotal *= 0.90
    return subtotal + taxa_entrega

precos = [39.90, 12.50, 8.00]
total = calcular_total(precos, 6.00, 50.00)
print(f"Total: R$ {total:.2f}")
```

Saída esperada

```
O cálculo fica separado da leitura e da apresentação.
```

Síntese do capítulo

- Projetos integram conceitos sem exigir uma aplicação grande.
- A função central deve ter um contrato testável.
- Regras de fronteira devem ser decididas e documentadas.

Projeto dois Analisador de notas

O segundo projeto recebe registros de estudantes e notas, calcula médias e produz classificações. Ele combina dicionários simples, percursos, funções, decisões e relatórios. O objetivo não é criar um sistema acadêmico completo, mas mostrar como uma especificação pode crescer sem perder organização.

Comece com registros definidos no próprio programa para eliminar ruído de entrada. Depois substitua a fonte por um arquivo de texto. Essa progressão permite testar o cálculo antes de investigar problemas de formato.

Use funções distintas para calcular uma média, classificar o resultado, formatar uma linha e gerar um resumo. Cada função deve ser testada com dados mínimos. Se o projeto apresentar um erro, será possível localizar se ele está na leitura, no cálculo ou na apresentação.

Um bom relatório não precisa ser extenso. Deve apresentar informações que respondam a uma pergunta: quantos foram aprovados, qual a média geral, quem obteve a maior média e quais registros foram rejeitados. A escolha da saída orienta as variáveis que precisam ser mantidas.

Tabela 8 — Possível organização do projeto

Parte	Responsabilidade	Saída
leitura	obter registros e validar campos	coleção de registros válidos
cálculo	produzir médias	valor numérico por estudante
classificação	aplicar faixas	situação acadêmica
relatório	agrupar e formatar	mensagens e indicadores

Síntese do capítulo

- Dados de exemplo permitem testar a lógica antes do arquivo.
- Uma função por responsabilidade reduz o custo da depuração.
- A pergunta do relatório determina o processamento necessário.

Projeto três Agenda de tarefas

Uma agenda simples representa tarefas com descrição, prioridade e estado. O programa oferece operações de adicionar, listar, concluir e remover. O exercício mostra como um menu pode ser apenas uma camada de interação sobre funções que trabalham com dados.

A primeira versão pode usar uma lista de dicionários. Não tente resolver ordenação avançada ou persistência antes de o fluxo básico funcionar. O importante é definir o que significa uma tarefa concluída, como uma tarefa é identificada e o que ocorre quando o identificador não existe.

A validação deve estar próxima da operação que conhece a regra. A função que conclui uma tarefa verifica se o identificador existe; a função que lê a opção verifica se o usuário escolheu um comando válido. Essa distribuição evita um bloco central com todas as decisões.

Como extensão, grave as tarefas em um arquivo ao sair e recarregue-as ao iniciar. Ao fazer isso, surgem decisões sobre formato, arquivo vazio e dados antigos. Essas decisões são oportunidades para aplicar o capítulo anterior.

Listagem de tarefas. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
tarefas = [
    {"id": 1, "texto": "ler capítulo", "concluida": False},
    {"id": 2, "texto": "resolver exercícios", "concluida": True},
]

for tarefa in tarefas:
    estado = "ok" if tarefa["concluida"] else "pendente"
    print(f'{tarefa["id"]}: {tarefa["texto"]} ({estado})')
```

Saída esperada

```
Cada registro é apresentado com seu estado.
```

Síntese do capítulo

- Menus são interfaces; regras devem viver em funções.
- Uma representação simples é melhor do que uma infraestrutura prematura.
- Persistência introduz novas decisões de formato e validação.

Roteiro de revisão final

Antes de considerar um programa concluído, releia o enunciado e compare cada regra com uma parte do código. Verifique se todas as entradas foram identificadas, se as conversões são coerentes, se as decisões cobrem os casos relevantes e se os laços terminam.

Depois revise os nomes. Uma pessoa que não participou da escrita consegue entender o papel de cada variável? Funções possuem contratos claros? Há uma função que concentra responsabilidades demais? A revisão de leitura costuma revelar problemas que a execução não mostra.

Execute testes comuns, de fronteira e inválidos. Registre resultados. Se houver arquivo ou entrada manual, repita o teste com dados diferentes. Um programa que funciona uma vez ainda não demonstrou estabilidade.

Por fim, remova mensagens temporárias, comentários obsoletos e código morto. Conserve explicações que ajudam o leitor e testes que preservam o comportamento. Qualidade não é quantidade de linhas: é a capacidade de explicar e verificar a solução.

Tabela 9 — Lista de verificação do projeto

Pergunta	Sim/Não
As entradas e unidades estão documentadas?	
Há validação para casos inválidos?	
Cada decisão possui testes?	
Os laços têm término demonstrável?	
As funções possuem responsabilidade clara?	
O resultado foi conferido independentemente?	

Síntese do capítulo

- Revisão compara especificação, implementação e testes.
- Nomes, contratos e separação de responsabilidades afetam a qualidade.
- Um programa concluído é um programa que pode ser explicado e verificado.

CAPÍTULO 10

Revisão geral e próximos passos

A lógica se fortalece quando o leitor explica a própria solução.

Mapa de conceitos

A progressão deste livro começou pela representação de problemas e chegou a pequenos projetos. O fio condutor foi o estado do programa: dados entram, expressões transformam valores, decisões escolhem caminhos, repetições atualizam o estado e funções organizam responsabilidades.

Python ofereceu uma sintaxe concreta para essas ideias. O conhecimento, entretanto, não está preso aos comandos apresentados. Outra linguagem pode usar palavras diferentes para blocos, tipos e funções, mas as perguntas de modelagem permanecem: que dados existem, que operações são necessárias e como comprovar o resultado?

O próximo passo é estudar estruturas de dados com maior profundidade. Listas, pilhas, filas, árvores, grafos e tabelas de espalhamento não são apenas recursos da biblioteca: são formas de organizar informação e de tornar certas operações mais adequadas. A escolha da estrutura modifica o algoritmo.

Continue praticando com problemas pequenos. Tente explicar uma solução sem mostrar o código, desenhe o estado em uma tabela e depois compare duas representações. A fluência nasce da alternância entre leitura, escrita, execução e revisão.

Síntese do capítulo

- Programas podem ser entendidos como transformações de estado.
- A lógica é transferível entre linguagens.
- Estruturas de dados aprofundam a relação entre representação e algoritmo.

CAPÍTULO 11

Caderno de laboratório

A prática guiada transforma conceitos em procedimentos que podem ser observados, testados e explicados.

Laboratório 1: cálculo incremental

Neste laboratório, o objetivo é decompor um cálculo em etapas nomeadas. Em vez de escrever uma fórmula extensa na saída, registre subtotal, acréscimo e total. A sequência de estados permite conferir cada parte com uma conta independente.

Use valores pequenos para que o resultado possa ser calculado manualmente. Depois altere a taxa e observe que a regra do programa continua a mesma. O exercício não é decorar uma fórmula: é aprender a tornar uma transformação visível.

Uma solução bem formada separa dados de entrada, processamento e apresentação. Se o programa precisar mudar a forma de exibir o total, o cálculo não deveria precisar ser reescrito. Essa separação é um primeiro passo em direção à modularização.

Registre também a unidade de cada valor. Misturar centavos e reais, minutos e horas ou porcentagem e fator decimal produz resultados que podem parecer plausíveis. Nomes e comentários curtos devem reduzir essa ambiguidade.

Cálculo em etapas. Depois de entender a versão básica, experimente uma entrada diferente.

```
preco = 80.0
quantidade = 3
taxa = 0.08

subtotal = preco * quantidade
acrescimo = subtotal * taxa
total = subtotal + acrescimo
print(f'Total: R$ {total:.2f}')
```

Saída esperada

```
Total: R$ 259.20
```

Ficha de cálculo

Anote as unidades e faça uma previsão antes de executar.

41. Calcule o consumo total de energia de três aparelhos.
42. Separe preço, desconto e frete em variáveis.
43. Teste quantidade zero e uma taxa igual a zero.
44. Explique qual variável mudaria se o frete fosse fixo.

Como conferir. Escolha um valor no limite da regra e confira o comportamento.

Síntese do capítulo

- Etapas nomeadas tornam o cálculo conferível.
- Unidades fazem parte do significado dos dados.
- A apresentação deve ficar separada da transformação.

Laboratório 2: validar antes de usar

Uma entrada externa é uma hipótese sobre o mundo, não um fato garantido. Antes de calcular uma divisão, por exemplo, verifique se o divisor é aceitável. Antes de classificar uma nota, verifique se ela está no domínio definido.

A validação deve produzir uma resposta clara. Um programa pode rejeitar o valor com uma mensagem, pedir nova entrada ou retornar um resultado especial. Em uma função reutilizável, retornar um status ou lançar uma exceção pode ser mais adequado do que imprimir diretamente.

Validações devem ser pequenas e testáveis. Se uma condição comprida mistura faixa, formato e regra comercial, extraia partes com nomes. O leitor consegue revisar cada afirmação e localizar qual hipótese falhou.

Teste tanto entradas inválidas quanto valores próximos do limite. A fronteira entre permitido e proibido é um lugar onde erros de desigualdade aparecem com frequência.

Validação em função. Preste atenção ao ponto em que o programa decide, repete ou transforma os dados.

```
def percentual(valor):
    if not 0 <= valor <= 100:
        raise ValueError('percentual fora do intervalo')
    return valor / 100

for valor in [0, 25, 100]:
    print(percentual(valor))
```

Saída esperada

```
0.0
0.25
1.0
```

Ficha de validação

Descreva o domínio antes de escrever a condição.

45. Valide uma idade não negativa.
46. Valide uma temperatura em uma faixa escolhida.
47. Valide um texto que não pode ser vazio.
48. Crie testes para os dois lados de cada limite.

Verificação. Procure uma inicialização que, se estivesse errada, alteraria o resultado.

Síntese do capítulo

- Validar é explicitar hipóteses sobre a entrada.
- Funções de validação isoladas são fáceis de testar.
- Limites precisam de casos específicos.

Laboratório 3: selecionar por tabela de regras

Quando há muitas faixas, escrever as regras em uma tabela antes do if ajuda a descobrir sobreposições e lacunas. Cada faixa deve indicar limite inferior, limite superior e resultado. A ordem do código deve refletir uma ordem compreensível na tabela.

Não trate o else como sinônimo de caso válido. Ele pode receber valores negativos, valores acima do máximo ou dados que não passaram por validação. Primeiro separe o domínio inválido; depois aplique a classificação.

Uma função de classificação deve receber um valor e devolver uma categoria. A camada de interação pode transformar a categoria em mensagem. Essa decisão permite testar a regra sem simular teclado.

Depois de implementar, crie uma tabela de decisão manual com pelo menos uma entrada em cada faixa. O teste de cobertura da regra é uma forma simples de evitar que uma categoria nunca seja alcançada.

Tabela 10 — Exemplo de faixas

Faixa	Categoria	Caso de teste
$x < 0$	inválido	-1
$0 \le x < 5$	inicial	0 e 4.9
$5 \le x < 8$	intermediário	5 e 7.9
$8 \le x \le 10$	avançado	8 e 10
$x > 10$	inválido	11

Classificação por faixas. Rode o exemplo uma vez e descreva, com suas palavras, o que aconteceu.

```
def nivel(pontuacao):
  if not 0 <= pontuacao <= 10:
    return 'inválido'
  if pontuacao < 5:
    return 'inicial'
  if pontuacao < 8:
    return 'intermediário'
  return 'avançado'
```

Saída esperada

```
A ordem das condições evita intervalos sobrepostos.
```

Ficha de decisão

Converta a tabela em código e preserve a ordem das faixas.

49. Classifique uma temperatura como fria, amena ou quente.
50. Classifique um valor de compra em faixas de desconto.
51. Verifique se alguma faixa ficou sem representação.
52. Inclua testes para cada limite.

Depois do teste. Compare uma versão simples com um caso um pouco maior.

Síntese do capítulo

- Tabelas revelam lacunas e sobreposições antes do código.
- Classificação pode ser separada da apresentação.
- Cada faixa precisa de um teste representativo.

Laboratório 4: contar e acumular

Contadores e acumuladores são variáveis que representam um resumo progressivo. O contador costuma começar em zero e aumentar em uma unidade; o acumulador começa no elemento neutro da operação. A inicialização depende da pergunta que está sendo respondida.

O laço deve indicar claramente quando um valor entra no resumo. Se o programa calcula a média, precisa de soma e quantidade; se encontra o maior valor, precisa tratar o primeiro item ou escolher um valor inicial seguro. A lista vazia é uma decisão de contrato.

Separar leitura, atualização e saída ajuda a rastrear o estado. Escreva uma tabela com iteração, entrada, acumulador e contador. Essa tabela mostra se a atualização ocorre antes ou depois da comparação.

Após o primeiro caso funcionar, teste zero itens, um item, valores repetidos e valores negativos quando o domínio permitir. Muitos erros em laços são erros de inicialização, não de sintaxe.

Resumo de uma sequência. Compare a saída com o contrato descrito no texto anterior.

```
valores = [8, 5, 11, 6]
soma = 0
maior = None

for valor in valores:
   soma += valor
   if maior is None or valor > maior:
      maior = valor
media = soma / len(valores)
print(media, maior)
```

Saída esperada

```
7.5 11
```

Ficha de repetição

Trace as variáveis após cada iteração.

53. Conte quantos valores são pares.
54. Calcule soma e média sem usar sum.
55. Encontre o menor valor sem ordenar.
56. Defina o que acontece para a lista vazia.

Para conferir. Registre a parte da solução que pode ser reutilizada em outro exercício.

Síntese do capítulo

- Inicialização vem da operação e do domínio.
- Tabela de rastreamento revela atualizações fora de ordem.
- Caso vazio precisa de uma decisão explícita.

Laboratório 5: repetição com sentinela

Uma sentinela é um valor reservado que indica o fim de uma sequência de entradas. Ela não é um dado comum da coleção. O programa precisa informar ao usuário a regra e verificar a sentinela antes de atualizar o resumo.

O padrão while com leitura no início evita processar a sentinela. Outra organização pode ler dentro do laço e interromper com break. As duas formas podem ser corretas; escolha a que deixa a condição de término mais fácil de localizar.

Uma sentinela inadequada cria ambiguidade. Se zero é um valor válido, não use zero para terminar. Quando não houver um valor reservado, uma quantidade conhecida, um arquivo ou uma interface explícita pode representar melhor o fim.

A repetição também precisa ser segura para entrada inválida. O capítulo sobre exceções mostra como tratar conversões; aqui, concentre-se em não perder a atualização de estado que controla o término.

Sentinela fora dos dados. Leia o código com uma entrada pequena e tente antecipar a saída.

```
soma = 0
quantidade = 0
valor = int(input('Valor (-1 termina): '))
while valor != -1:
    soma += valor
    quantidade += 1
    valor = int(input('Valor (-1 termina): '))

media = soma / quantidade if quantidade else 0
print(media)
```

Saída esperada

```
A média é zero quando nenhum valor foi informado.
```

Ficha de sentinela

Escolha a sentinela depois de listar valores válidos.

57. Leia temperaturas até a palavra fim.
58. Some preços até receber um código de término.
59. Explique por que uma sentinela numérica pode ser ambígua.
60. Trate a entrada vazia como uma sequência sem itens.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- A sentinela deve ser distinguível dos dados válidos.
- O teste de término deve ocorrer no ponto certo.
- Fim de entrada e sequência vazia são casos diferentes.

Laboratório 6: desenhar padrões com laços

Laços aninhados aparecem quando uma repetição deve ser executada para cada item de outra repetição. Uma tabela de multiplicação, uma grade ou a enumeração de pares são exemplos. O laço externo escolhe a linha ou o primeiro elemento; o interno completa a combinação.

Antes de executar, desenhe uma pequena grade e anote a ordem de visita. A ordem ajuda a decidir onde colocar a saída, o acumulador e a reinicialização. Um acumulador que deveria começar a cada linha não pode ficar criado apenas uma vez antes dos dois laços.

O custo de dois laços não é automaticamente $O(n^2)$: depende da quantidade de vezes que cada um executa. Se um laço percorre n e o outro percorre uma constante, o crescimento é linear. Conte as iterações no caso de teste.

Em exercícios de padrão, prefira construir uma linha por vez e só depois apresentá-la. Essa estratégia separa o conteúdo da linha da forma como a linha é impressa.

Grade construída por linhas. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
for linha in range(1, 4):
    partes = []
    for coluna in range(1, 4):
        partes.append(str(linha * coluna))
    print(' '.join(partes))
```

Saída esperada

```
1 2 3
2 4 6
```

```
3 6 9
```

Ficha de laços aninhados

Conte as iterações para uma grade 4 por 4.

61. Imprima uma tabela de adição.
62. Liste todos os pares de uma sequência.
63. Conte quantas combinações satisfazem uma condição.
64. Separe a construção da linha da impressão.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Laço externo e interno possuem papéis diferentes.
- Reinicialização deve ocorrer no nível correto.
- Contar iterações ajuda a estimar o custo.

Laboratório 7: transformar em função

Neste laboratório, transforme um script monolítico em funções pequenas. Uma função deve responder a uma pergunta ou executar uma responsabilidade. Quando recebe parâmetros e devolve resultado, pode ser testada com valores definidos pelo próprio programa.

Comece marcando as etapas do script: ler, converter, calcular, classificar e apresentar. As duas primeiras e a última dependem do ambiente; o cálculo e a classificação podem ser funções puras. A divisão não precisa ser perfeita na primeira tentativa.

Uma função bem nomeada substitui detalhes por uma intenção. O chamador lê calcular_total e entende a operação sem precisar conhecer cada multiplicação. A implementação pode mudar depois sem alterar a chamada.

Evite retornar e imprimir o mesmo resultado sem necessidade. Defina se a função produz um valor ou um efeito. Misturar os dois reduz a possibilidade de reutilização e torna os testes mais difíceis.

Função que devolve um resultado. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
def calcular_total(precos, desconto):
    subtotal = sum(precos)
    return subtotal * (1 - desconto)

precos = [20, 15, 5]
print(calcular_total(precos, 0.10))
```

Saída esperada

```
36.0
```

Ficha de modularização

Desenhe as responsabilidades antes de mover o código.

65. Extraia uma função para calcular média.
66. Extraia uma função para validar nota.
67. Teste cada função sem input.
68. Identifique uma variável que deveria ser parâmetro.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- Funções reduzem a quantidade de decisões simultâneas.
- Parâmetros e retornos formam um contrato.
- Cálculos puros podem ser testados sem interação.

Laboratório 8: percorrer textos

Strings são sequências e podem ser percorridas com for. O problema de contar caracteres permite praticar estado, condição e dicionário. Antes de programar, defina se espaços, pontuação e diferenças entre maiúsculas e minúsculas serão considerados.

A normalização deve ser uma etapa identificável. Converter para minúsculas não é o mesmo que remover acentos ou pontuação. Se a regra não for definida, duas pessoas podem produzir contagens diferentes para o mesmo texto.

Ao construir uma nova string, lembre-se de que strings são imutáveis. Operações como replace devolvem uma nova sequência. Em textos longos, juntar partes em uma lista e usar join ao final costuma expressar melhor a intenção.

Teste frases vazias, com uma palavra, com repetição e com caracteres fora do alfabeto escolhido. A função deve deixar claro se recebe texto bruto ou texto previamente normalizado.

Contagem de caracteres. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
def contar_letras(texto):
    contagem = {}
    for caractere in texto.lower():
        if caractere.isalpha():
            contagem[caractere] = contagem.get(caractere, 0) + 1
    return contagem

print(contar_letras('Ana!'))
```

Saída esperada

```
{'a': 2, 'n': 1}
```

Ficha de strings

Escreva a política de normalização no início da solução.

69. Conte vogais e consoantes.
70. Retorne a primeira posição de uma letra.
71. Remova espaços duplicados.
72. Compare uma frase com sua versão invertida.

Para conferir. Altere os valores do exemplo e confirme se a regra continua válida.

Síntese do capítulo

- String é uma sequência que pode ser percorrida.
- Normalização altera a pergunta feita aos dados.
- Imutabilidade orienta a construção de novos textos.

Laboratório 9: escolher uma coleção

Uma mesma informação pode ser representada por lista, conjunto ou dicionário. Use uma lista quando posição e repetição importarem; um conjunto quando a pergunta for de pertencimento e unicidade; um dicionário quando uma chave precisar localizar um valor.

Escolha a estrutura a partir das operações, não de uma preferência estética. Se o programa precisa saber rapidamente se um código já apareceu, um conjunto expressa melhor a regra. Se precisa devolver o nome associado a um código, um dicionário torna a relação explícita.

Conversões entre coleções podem ser úteis, mas têm custo e podem perder informação. Transformar uma lista em conjunto remove repetições e não deve ser feito sem que essa perda seja aceitável. Ordenar um conjunto também cria uma nova sequência para apresentação.

Descreva a representação com uma frase útil para quem fará manutenção. Explique o papel de cada posição, chave ou elemento; repetir o nome do tipo não basta.

Tabela 11 — Pergunta e coleção

Pergunta	Estrutura inicial	Motivo
O item ocupa uma posição?	lista	acesso e ordem
O item pode aparecer uma vez?	set	unicidade e pertencimento
Qual valor corresponde à chave?	dict	associação
A ordem de entrada deve ser preservada?	list ou dict	sequência observável

Três representações. Faça uma previsão, execute e compare os dois resultados.

```
nomes = ['ana', 'bia', 'ana']
unicos = set(nomes)
telefones = {'ana': '5551', 'bia': '5552'}
print(len(nomes), len(unicos), telefones['ana'])
```

Saída esperada

```
3 2 5551
```

Ficha de representação

Para cada problema, nomeie a operação mais frequente.

73. Mantenha alunos matriculados sem repetição.
74. Associe cada aluno à sua nota.
75. Preserve a ordem de leitura de um arquivo.
76. Justifique uma conversão entre tipos.

Como conferir. Releia o enunciado e verifique se cada condição foi contemplada.

Síntese do capítulo

- Estruturas expressam perguntas frequentes.
- Conversão pode remover ordem ou repetição.
- A representação deve ser documentada.

Laboratório 10: ler um arquivo com segurança

Arquivos podem não existir, estar vazios ou conter linhas fora do formato esperado. O programa precisa definir abertura, codificação e resposta para cada situação. Inclua essas decisões no contrato, não as deixe escondidas na implementação.

Usar with open garante que o arquivo seja fechado ao terminar o bloco. O parâmetro encoding explicita como os bytes serão interpretados. Em exercícios, utf-8 é uma escolha comum; em sistemas reais, a codificação deve acompanhar a origem dos dados.

Uma função de leitura pode devolver linhas limpas, enquanto outra interpreta cada linha. Separar as responsabilidades facilita testar a interpretação com strings sem criar arquivos temporários.

Não esconda erros de leitura com um except amplo que não informa nada. Uma mensagem deve ajudar a identificar o caminho e a operação que falhou. Também é possível tratar arquivo ausente como coleção vazia, se essa for a regra.

Leitura com fechamento automático. Observe como a entrada passa por cada etapa até chegar à saída.

```
def ler_nomes(caminho):
    with open(caminho, encoding='utf-8') as arquivo:
        return [linha.strip() for linha in arquivo if linha.strip()]

nomes = ler_nomes('nomes.txt')
print(len(nomes))
```

Saída esperada

```
A função devolve somente linhas não vazias.
```

Ficha de arquivos

Crie um arquivo pequeno e compare o conteúdo lido.

77. Leia números, convertendo uma linha por vez.
78. Conte linhas vazias e não vazias.
79. Grave um relatório em outro arquivo.
80. Defina o comportamento para caminho inexistente.

Verificação. Acompanhe o contador ou acumulador em uma tabela curta.

Síntese do capítulo

- Arquivo é uma entrada que pode falhar.
- with open administra o fechamento do recurso.
- Leitura e interpretação podem ser separadas.

Laboratório 11: tratar exceções

Exceções informam que uma operação não completou normalmente. O tratamento deve ficar próximo do ponto em que o programa sabe como agir. Capturar ValueError pode permitir pedir outro número; capturar FileNotFoundError pode apresentar uma orientação sobre o caminho.

Uma exceção não é sempre um defeito. O usuário pode ter digitado texto quando um número era esperado; um arquivo pode realmente não existir. O programa precisa decidir quais situações são esperadas e quais devem continuar visíveis para investigação.

Use try para uma região pequena. Se colocar todo o programa dentro do bloco, fica difícil saber qual operação falhou. A mensagem apresentada deve ser diferente da informação técnica registrada durante a depuração.

Depois de tratar, o estado precisa continuar coerente. Se uma conversão falhar, não atualize o acumulador com um valor inexistente. Se uma inserção falhar, não anuncie que o item foi incluído.

Repetir uma leitura inválida. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

```
def ler_inteiro(mensagem):
    while True:
        try:
            return int(input(mensagem))
        except ValueError:
            print('Digite um número inteiro.')

quantidade = ler_inteiro('Quantidade: ')
print(quantidade)
```

Saída esperada

```
A função só retorna depois de obter um inteiro.
```

Ficha de exceções

Liste a operação que pode falhar antes de escrever try.

81. Trate divisão por zero com mensagem adequada.
82. Trate número fora do intervalo.
83. Trate arquivo ausente.
84. Explique por que except Exception é amplo demais.

Depois do teste. Teste a entrada mínima, vazia ou ausente quando ela fizer sentido.

Síntese do capítulo

- Trate exceções onde existe uma decisão possível.
- Blocos try pequenos produzem mensagens melhores.
- O estado só deve mudar após a operação bem-sucedida.

Laboratório 12: reproduzir e corrigir um erro

Depurar é investigar uma diferença entre comportamento esperado e observado. Comece escrevendo o menor exemplo que ainda falha. Um caso pequeno reduz o número de estados e torna a hipótese mais específica.

Observe os valores em pontos estratégicos, não em todas as linhas. Um print antes de uma atualização e outro depois podem revelar se o problema está na entrada, no cálculo ou na saída.

Formule uma hipótese antes de editar. Se a hipótese for que o contador é atualizado duas vezes, altere apenas esse trecho e observe se o sintoma previsto desaparece. Alterações sem registro impedem saber qual mudança ajudou.

Depois da correção, mantenha um teste que falharia com o defeito antigo. Assim, a investigação se transforma em proteção para o futuro.

Teste do caso vazio. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
def media(valores):
    if not valores:
        return None
    total = 0
    for valor in valores:
        total += valor
    return total / len(valores)

assert media([2, 4]) == 3
```

```
assert media([]) is None
```

Saída esperada

```
Os testes tornam o contrato visível.
```

Ficha de depuração

Registre sintoma, hipótese, evidência e correção.

85. Introduza um erro de limite e encontre-o.
86. Corrija um acumulador inicializado com valor inadequado.
87. Use assert para preservar uma propriedade.
88. Escreva o teste mínimo para a falha.

Para conferir. Explique em uma frase por que o resultado atende ao enunciado.

Síntese do capítulo

- Um caso mínimo torna a causa mais fácil de observar.
- Hipóteses orientam mudanças pequenas.
- Teste de regressão preserva a correção.

Laboratório 13: refatorar sem alterar a regra

Refatorar é melhorar a organização interna preservando o comportamento observável. O primeiro passo é registrar exemplos de entrada e saída. Sem essa referência, uma mudança de nomes ou funções pode alterar a regra sem que o autor perceba.

Uma refatoração segura pode extrair função, renomear variável, reduzir duplicação ou separar interação de cálculo. Faça uma mudança por vez e execute os testes depois. A sequência curta mantém a última versão compreensível.

Código duplicado convida a divergências: uma regra é corrigida em um lugar e esquecida em outro. Por outro lado, abstrair cedo demais pode criar funções genéricas que escondem a intenção. Extraia quando houver responsabilidade clara ou repetição real.

A leitura final deve ser feita como se você fosse uma pessoa nova no projeto. Os nomes explicam o domínio? A função principal mostra o fluxo? Os detalhes estão em funções que podem ser conferidas separadamente?

Uma regra compartilhada. Depois de entender a versão básica, experimente uma entrada diferente.

```
def aplicar_desconto(preco, percentual):
    return preco * (1 - percentual)

def totalizar(precos, percentual):
    return sum(aplicar_desconto(p, percentual) for p in precos)
```

Saída esperada

```
A função de desconto evita repetir a fórmula.
```

Ficha de refatoração

Guarde resultados antigos antes de reorganizar.

89. Extraia uma expressão repetida para uma função.
90. Renomeie variáveis vagas.
91. Separe input de uma função de cálculo.
92. Explique qual comportamento foi preservado.

Como conferir. Escolha um valor no limite da regra e confira o comportamento.

Síntese do capítulo

- Exemplos e testes protegem uma refatoração.
- Abstração deve revelar uma responsabilidade real.
- Separar interação de cálculo melhora a reutilização.

Laboratório 14: projeto de consolidação

Escolha um problema que exija entrada, decisão, repetição, função e coleção. Sugestões incluem controle de despesas, analisador de notas ou agenda de tarefas. Comece com uma versão mínima que resolva um único fluxo completo.

Escreva o contrato em uma página: dados de entrada, formato, operações, mensagens, casos inválidos e resultado. Em seguida, faça uma tabela com exemplos. A tabela funciona como ponte entre o enunciado e os testes.

Implemente em camadas. Primeiro escreva funções de transformação com dados fixos; depois adicione leitura; por fim organize a saída. Se o programa ficar difícil de ler, volte à lista de responsabilidades e extraia a parte que concentra decisões.

A entrega deve incluir uma explicação da representação, uma análise simples de custo e uma lista de limitações. Dizer o que a solução ainda não trata demonstra compreensão do escopo.

Tabela 12 — Roteiro de entrega

Parte	Pergunta orientadora
Problema	Que pergunta o programa responde?
Dados	Quais tipos e unidades existem?
Algoritmo	Quais passos transformam a entrada?
Código	Quais funções expressam responsabilidades?
Testes	Que casos confirmam a regra?
Limites	O que ainda não está coberto?

Projeto final do laboratório

Produza uma versão executável e uma página de documentação.

93. Inclua ao menos cinco testes.
94. Inclua um caso inválido e um caso de fronteira.
95. Explique uma decisão de representação.
96. Faça uma pequena refatoração após os testes.

Verificação. Procure uma inicialização que, se estivesse errada, alteraria o resultado.

Síntese do capítulo

- Projetos integram os elementos da lógica de programação.

- Contrato e exemplos orientam a implementação.
- Limitações documentadas tornam a entrega honesta e útil.

CAPÍTULO 12

Banco de desafios e revisão

Resolver problemas variados revela quais ideias já podem ser usadas sem apoio.

Desafios graduados

Use esta seção como avaliação formativa. Tente resolver cada desafio sem olhar as respostas orientativas. Para cada solução, registre a representação dos dados, o algoritmo, os casos de teste e uma frase sobre o custo.

Os primeiros desafios podem ser resolvidos com expressões e decisões. Os seguintes exigem repetição, coleções e funções. Os últimos pedem combinação de ideias e explicação das escolhas. A dificuldade está na precisão da especificação, não apenas na quantidade de código.

Quando não souber começar, reduza a entrada para um exemplo com um ou dois elementos. Resolva manualmente, observe qual informação precisa permanecer na memória e transforme essa informação em variável ou coleção. Depois generalize.

Uma solução diferente da resposta orientativa pode estar correta. O critério é a correspondência com o contrato, não a semelhança textual. Compare casos de fronteira e justifique qualquer diferença de interpretação.

Desafios

Entregue código, testes e uma explicação breve para cada item.

97. Calcule o troco usando a menor quantidade de cédulas para valores inteiros.
98. Encontre a maior sequência crescente consecutiva.
99. Remova palavras repetidas preservando a primeira ocorrência.
100. Leia registros e retorne a categoria com maior média.
101. Crie uma função que converta segundos em uma mensagem legível.
102. Simule uma fila de atendimento com prioridade de emergência.
103. Faça um menu que encerre somente com uma opção de saída.
104. Compare duas listas e apresente itens comuns e exclusivos.
105. Implemente uma função que valide uma senha segundo regras dadas.
106. Documente um pequeno programa escrito em capítulo anterior.

Depois do teste. Compare uma versão simples com um caso um pouco maior.

Síntese do capítulo

- Especificação, exemplo e teste formam um ciclo de aprendizagem.
- Problemas novos podem ser reduzidos a casos pequenos.
- Correção inclui justificar escolhas e limites.

Rubrica de autoavaliação

Avalie cada projeto em uma escala de 0 a 2: 0 quando o item não aparece, 1 quando aparece parcialmente e 2 quando está claro e verificável. A rubrica não substitui avaliação externa; ela ajuda a localizar o próximo estudo.

Uma pontuação baixa em lógica não deve ser corrigida com mais sintaxe. Volte ao enunciado, escreva exemplos e desenhe o estado. Uma pontuação baixa em testes pede casos de fronteira e comparação entre resultado previsto e observado.

Revise também a comunicação. Um código correto, mas impossível de explicar, ainda precisa de nomes melhores, funções menores ou comentários de intenção. Programar em equipe exige que a solução sobreviva à leitura de outra pessoa.

Ao terminar, escolha dois itens para melhorar e registre uma ação concreta: reescrever função, criar três testes, explicar decisão ou comparar duas estruturas. Aprender é transformar observação em próxima tentativa.

Tabela 13 — Rubrica

Critério	0	1	2
Modelo do problema	ausente	parcial	explícito
Fluxo de controle	confuso	funciona em casos	caminhos claros
Dados e tipos	inadequados	parcialmente definidos	coerentes
Testes	não há	casos comuns	fronteiras e inválidos
Organização	monolítica	alguma separação	funções com contrato
Explicação	não justifica	descreve passos	justifica escolhas e limites

Síntese do capítulo

- Autoavaliação deve indicar ações de melhoria.
- Clareza e testes fazem parte da competência de programação.
- O próximo estudo nasce dos erros observados.

CAPÍTULO 13

Guia de estudo e revisão

A autonomia nasce quando o estudante consegue explicar, testar e revisar uma solução.

Ler pseudocódigo

Pseudocódigo não é uma linguagem com uma sintaxe única. Ele é uma representação intermediária que deve deixar claro quais dados entram, como o estado muda e quando o algoritmo termina. A liberdade de escrita não autoriza omitir decisões relevantes.

Ao ler, marque verbos de entrada, atribuições, decisões, repetições e saída. Depois traduza cada marca para uma operação mental: ler um valor, atualizar uma variável, escolher um caminho, repetir um bloco ou apresentar um resultado.

Um bom pseudocódigo pode ser revisado por alguém que não conheça Python. Se uma instrução exige adivinhar o tipo de um dado ou o limite de um laço, a representação ainda está incompleta.

Use a passagem para Python como uma etapa de tradução. Não mude a regra durante a tradução sem registrar a mudança; caso contrário, não será possível saber se um resultado diferente veio da linguagem ou do algoritmo.

Pseudocódigo com repetição conhecida. Preste atenção ao ponto em que o programa decide, repete ou transforma os dados.

```
algoritmo media_positivos
  leia quantidade
  soma <- 0
  repita quantidade vezes
    leia valor
    soma <- soma + valor
  fim repita
  escreva soma / quantidade
fim algoritmo
```

Saída esperada

```
A tradução precisa decidir como tratar quantidade igual a zero.
```

Oficina de tradução

Escreva primeiro em português estruturado e só depois em Python.

107. Descreva o algoritmo de uma conversão de unidades.
108. Indique a entrada e a saída de um classificador.
109. Marque a condição de término de uma repetição.
110. Identifique uma decisão ausente no pseudocódigo.

Para conferir. Registre a parte da solução que pode ser reutilizada em outro exercício.

Síntese do capítulo

- Pseudocódigo deve ser preciso sem depender da sintaxe.
- Ler é localizar estado, caminhos e término.
- A tradução para Python não deve esconder mudanças de regra.

Ler fluxogramas

Um fluxograma oferece uma visão espacial do controle. Retângulos representam ações, losangos representam decisões e setas indicam o próximo passo. O desenho ajuda a conversar sobre caminhos antes que eles sejam escritos em uma linguagem.

Siga uma seta com uma entrada concreta e escreva o valor das variáveis ao lado de cada ação. Se duas setas se juntam, verifique se ambas chegam a um estado compatível. Se uma seta não chega a lugar algum, há uma saída ou um erro de modelagem.

Fluxogramas ficam difíceis quando cada detalhe da implementação é desenhado. Use o nível de abstração adequado à pergunta. Para discutir uma regra, mostre decisões principais; para depurar um laço, detalhe a condição e a atualização.

O desenho também pode revelar caminhos impossíveis, ciclos sem saída e saídas que não informam o resultado. Corrigir o fluxograma antes do código costuma ser mais barato do que corrigir um programa já cheio de detalhes.

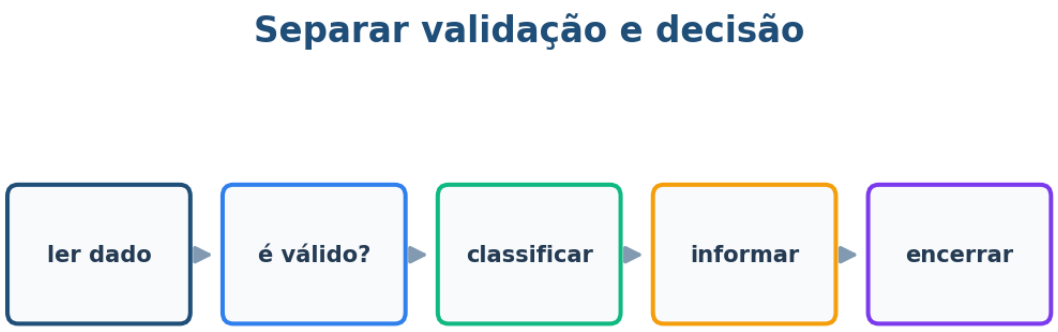


Figura 7 — Caminhos de um algoritmo

Oficina de fluxogramas

Desenhe usando poucos símbolos e rotule as saídas.

111. Represente uma média com validação.
112. Represente um laço que lê até sentinela.
113. Marque um caminho de erro.
114. Compare o desenho com o código equivalente.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- Setas representam transições de controle.
- Entradas concretas ajudam a testar o caminho.
- Desenhos devem manter um nível de abstração legível.

Tabelas de rastreamento

Uma tabela de rastreamento registra o estado do programa depois de cada passo relevante. As colunas podem conter iteração, entrada, contador, acumulador e saída parcial. Ela transforma uma execução invisível em uma sequência que pode ser conferida.

Não é necessário registrar toda expressão. Escolha variáveis que participam da decisão ou do resultado. Se o valor não muda e não influencia o caminho, ele provavelmente não precisa de uma coluna.

Compare a tabela prevista com a tabela produzida pelo programa. A primeira diferença é uma pista forte: antes dela, o estado ainda estava coerente; depois dela, alguma atualização ou condição precisa ser investigada.

Tabelas também servem para explicar algoritmos a outra pessoa. Ao apresentar uma busca ou um acumulador, mostre duas ou três linhas da evolução e descreva a regra que se repete.

Tabela 14 — Rastreamento de uma soma

Passo	valor	soma antes	soma depois
início	—	0	0
1	4	0	4
2	7	4	11
3	2	11	13

Código para conferir a tabela. Rode o exemplo uma vez e descreva, com suas palavras, o que aconteceu.

```
soma = 0
for valor in [4, 7, 2]:
    soma += valor
    print(valor, soma)
```

Saída esperada

```
4 4
7 11
2 13
```

Oficina de rastreamento

Registre apenas variáveis que alteram a decisão ou o resultado.

115. Trace um contador de pares.
116. Trace a busca do maior valor.
117. Trace uma classificação por faixas.
118. Compare a primeira linha divergente.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Rastreamento explicita mudanças de estado.
- A primeira divergência reduz o espaço de investigação.
- Tabelas são úteis para aprender e comunicar.

Projetar casos de teste

Um teste é uma pergunta com resultado esperado. O conjunto de testes deve representar o comportamento do domínio, não apenas exercitar linhas de código. Para cada regra, procure um caso comum, uma fronteira e uma entrada inválida quando ela fizer sentido.

Testes de fronteira são especialmente importantes para comparações. Se o domínio aceita idade maior ou igual a 18, inclua 17, 18 e 19. Esses valores revelam se o sinal da desigualdade e a inclusão do limite foram implementados corretamente.

Para funções que devolvem coleções, verifique conteúdo, ordem e repetição conforme o contrato. Para funções que alteram uma estrutura, verifique o estado antes e depois. Para funções que levantam exceção, verifique o tipo da exceção e a mensagem útil.

Um teste que falha deve ser pequeno e informativo. Dê um nome que descreva a regra, não o mecanismo. Nomes como test_nota_no_limite ajudam mais do que test_3.

Tabela 15 — Categorias de teste

Categoria	Exemplo de pergunta	Por que existe
comum	o fluxo principal funciona?	confirma o uso esperado
fronteira	o limite é inclusivo?	revela erros de comparação
inválido	o domínio rejeita o dado?	verifica proteção
vazio	qual resultado sem itens?	define ausência
repetido	a duplicação é preservada?	confirma a semântica

Teste de uma fronteira. Compare a saída com o contrato descrito no texto anterior.

```
def aprovado(nota):
  if not 0 <= nota <= 10:
    raise ValueError('nota inválida')
  return nota >= 7

assert aprovado(7) is True
assert aprovado(6.99) is False
```

Saída esperada

```
O valor 7 documenta que o limite é inclusivo.
```

Oficina de testes

Para cada regra, escreva entradas e resultados antes do código.

119. Crie testes para uma função de desconto.
120. Inclua lista vazia e lista unitária.
121. Teste texto com repetição.
122. Dê nomes descritivos aos testes.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- Testes representam regras e não apenas linhas.

- Fronteiras e casos vazios merecem atenção.
- Nomes de teste documentam o comportamento.

Invariantes de laços

Um invariante é uma propriedade que permanece verdadeira no início ou no fim de cada iteração. Em um acumulador, pode ser que a variável contenha a soma dos itens já processados. Em uma busca, pode ser que todos os itens antes do índice atual já tenham sido examinados.

Escrever o invariante ajuda a decidir a inicialização, a atualização e o momento de testar a condição. Também oferece uma explicação para a correção: se a propriedade começa verdadeira e a iteração a preserva, ela continua verdadeira quando o laço termina.

Não é preciso usar uma prova formal para começar. Escreva em linguagem natural o que a variável significa depois de uma, duas e três iterações. Se a frase muda de sentido, o estado está mal definido.

A condição de término completa a argumentação. O invariante diz o que já foi processado; o término diz que não resta parte relevante, ou que a busca concluiu ausência. As duas ideias devem aparecer juntas.

Invariante de uma busca. Leia o código com uma entrada pequena e tente antecipar a saída.

```
def contem(valores, alvo):
    indice = 0
    while indice < len(valores):
        if valores[indice] == alvo:
            return True
        indice += 1
    return False
```

Saída esperada

```
Antes de cada iteração, posições menores que indice já foram examinadas.
```

Oficina de invariantes

Escreva uma frase começando com: já processei...

123. Descreva o invariante de uma soma.
124. Descreva o invariante de um contador.
125. Explique o que o término garante.
126. Encontre um laço cuja variável não avança.

Para conferir. Altere os valores do exemplo e confirme se a regra continua válida.

Síntese do capítulo

- Invariante dá significado ao estado durante a repetição.
- Inicialização e atualização devem preservar a frase.
- Término explica o que ainda não resta processar.

Complexidade sem formalismo excessivo

Complexidade pergunta como o trabalho cresce quando a entrada aumenta. Em uma busca linear, dobrar a quantidade de itens pode aproximadamente dobrar o número de comparações. Em uma busca binária, o intervalo cai pela metade a cada passo.

Para fazer uma estimativa inicial, conte a operação dominante e ignore detalhes constantes. Dois laços independentes que percorrem n itens continuam lineares; um laço dentro de outro pode ser quadrático quando ambos percorrem n.

A notação não descreve tempo exato em segundos. Ela permite comparar tendências e revelar gargalos prováveis. Uma implementação O(n) pode ser mais lenta que uma O(n log n) em um tamanho pequeno, dependendo das constantes e da linguagem.

Analise a operação que a aplicação realmente repete. Ordenar uma vez e buscar muitas vezes pode ser melhor do que fazer uma busca linear em cada consulta. A organização dos dados faz parte do algoritmo.

Tabela 16 — Padrões de crescimento

Padrão	Exemplo	Ordem aproximada
uma passagem	percorrer todos os itens	O(n)
metade por passo	busca binária	O(log n)
duas dimensões	comparar todos os pares	$O(n^2)$
ordenar e percorrer	ordenação eficiente	O(n log n)

Oficina de custo

Conte iterações para n igual a 10 e depois generalize.

127. Analise uma busca por nome em lista.
128. Analise a construção de uma tabela de frequências.
129. Compare duas passagens independentes.
130. Explique uma constante que a notação ignora.

Como conferir. Releia o enunciado e verifique se cada condição foi contemplada.

Síntese do capítulo

- Complexidade descreve crescimento, não segundos exatos.
- Conte a operação dominante do uso real.
- Organizar dados pode mudar o custo das consultas.

Revisar código de outra pessoa

Uma revisão útil começa entendendo o contrato, não corrigindo estilo. Execute um exemplo, leia os nomes e localize o fluxo principal. Depois procure entradas não tratadas, decisões sobrepostas, laços sem atualização e funções com responsabilidades misturadas.

Comente o risco e a pergunta que precisa ser respondida. Em vez de dizer que o código está ruim, indique: o que acontece para uma lista vazia? A unidade está documentada? O laço termina quando a entrada é inválida? Perguntas acionáveis tornam a revisão colaborativa.

Distingua defeito de preferência. Espaços, nomes e formatação podem seguir convenções; uma saída incorreta ou uma exceção inesperada é um problema de comportamento. Priorize o que ameaça correção, segurança, desempenho ou manutenção.

Depois da revisão, reconheça o que já está claro. Uma função bem isolada ou um teste de fronteira pode ser preservado durante a mudança. Revisar também é aprender estratégias de outras pessoas.

Oficina de revisão

Use perguntas concretas e proponha um teste para cada risco.

131. Revise uma função que calcula média.
132. Revise um laço com sentinela.
133. Revise um menu de tarefas.
134. Separe preferência de defeito.

Verificação. Acompanhe o contador ou acumulador em uma tabela curta.

Síntese do capítulo

- Revisão começa pelo contrato e pelo comportamento.
- Perguntas concretas produzem mudanças verificáveis.
- Priorize correção e mantenha boas decisões.

Plano de estudo

Um plano sustentável alterna leitura, execução, escrita e revisão. Reserve um bloco para compreender o conceito, outro para digitar e alterar exemplos e outro para resolver um problema sem apoio. O intervalo entre tentativas ajuda a consolidar o raciocínio.

Mantenha um caderno de padrões: como validar, como acumular, como percorrer, como separar funções e como testar fronteiras. Não cole apenas código; escreva a pergunta que cada padrão responde e o limite que ele possui.

A cada semana, escolha um programa antigo e faça uma pequena melhoria. Pode ser adicionar testes, explicar tipos, reduzir duplicação ou tratar uma entrada. Melhorias incrementais criam repertório de revisão.

Ao concluir este livro, o próximo passo pode ser estruturas de dados, orientação a objetos, análise de algoritmos, banco de dados ou engenharia de software. Leve a mesma disciplina: modelo, contrato, implementação, teste e reflexão.

Tabela 17 — Ciclo semanal

Momento	Atividade	Evidência
compreender	ler e resumir	mapa de conceitos
praticar	executar e modificar	diário de observações
resolver	fazer desafio sem apoio	código e testes
revisar	explicar e refatorar	mudança justificada

Plano pessoal

Escolha uma prática para repetir nas próximas quatro semanas.

135. Defina dois dias de prática.
136. Escolha um projeto pequeno.
137. Registre três erros e suas hipóteses.
138. Escreva uma meta observável para a próxima revisão.

Depois do teste. Teste a entrada mínima, vazia ou ausente quando ela fizer sentido.

Síntese do capítulo

- Aprendizagem combina leitura, execução e explicação.
- Padrões devem ser entendidos junto com seus limites.
- O próximo assunto aproveita o mesmo ciclo de trabalho.

CAPÍTULO 14

Projeto guiado organizador de estudos

Um programa completo nasce de um contrato pequeno, testável e progressivamente ampliado.

Especificar antes de programar

Objetivos de aprendizagem. Ao concluir esta parte, você deverá conseguir:

- transformar um enunciado cotidiano em entradas, regras e saídas
- usar dicionários e listas para representar registros
- decompor o programa em funções testáveis
- validar dados e discutir limites de uma solução

O projeto deste capítulo é um organizador de estudos. O programa recebe disciplinas, quantidade de horas disponíveis e prioridades; em seguida, produz uma sugestão de ordem de estudo. O objetivo não é criar um planejador universal, mas praticar a passagem de uma necessidade cotidiana para um conjunto de regras explícitas.

Uma especificação mínima precisa dizer o que entra, o que sai e o que não será decidido automaticamente. Neste exemplo, o programa não conhece o calendário real, não promete uma agenda ótima e não substitui a decisão do estudante. Ele apenas ordena informações segundo critérios que podem ser examinados.

Comece pelo caso pequeno. Com duas disciplinas, é possível calcular manualmente a pontuação e conferir cada etapa. Se a solução não for explicável para duas entradas, acrescentar vinte entradas apenas esconde o problema. O tamanho maior deve testar o algoritmo, não substituir a compreensão.

Tabela 18 — Contrato do organizador

Elemento	Decisão do projeto	Exemplo
entrada	lista de disciplinas e horas semanais	Algoritmos, 4 horas
regra	prioridade maior aparece antes	prova próxima
saída	lista ordenada com justificativa	estudar primeiro
limite	não calcula uma agenda ótima	não considera deslocamento

Síntese do capítulo

- Especificação separa o que o programa faz do que ele não promete.
- Casos pequenos permitem conferir a regra manualmente.
- Entradas, transformação e saída formam o contrato inicial.

Modelar registros com dicionários

Cada disciplina pode ser representada por um dicionário com nome, horas e prioridade. A lista preserva a coleção; o dicionário dá nome aos campos. Essa combinação é suficiente para uma primeira versão e torna o programa legível para quem ainda está aprendendo classes.

Escolha nomes que expressem a unidade. horas_semanais é diferente de horas_totais; prioridade pode ser uma escala definida pelo projeto. Quando uma unidade ou faixa não estiver clara, o dado ainda não está pronto para o cálculo.

A função pontuar não precisa conhecer a forma como os dados foram lidos. Ela recebe um registro e devolve um número. Separar cálculo de interação permite testar a regra com uma lista literal, sem depender de input ou arquivo.

Registros nomeados. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
disciplinas = [
  {"nome": "Algoritmos", "horas": 4, "prioridade": 3},
  {"nome": "Redes", "horas": 2, "prioridade": 1},
]

def pontuar(disciplina):
  return disciplina["prioridade"] * 10 + disciplina["horas"]

print(pontuar(disciplinas[0]))
```

Saída esperada

```
34
```

Ficha de modelagem

Escreva o significado de cada campo antes de ordenar.

139. Acrescente uma data de avaliação.
140. Inclua uma unidade para a carga de estudo.
141. Defina o que acontece quando falta um campo.
142. Explique por que a lista e o dicionário têm papéis diferentes.

Para conferir. Explique em uma frase por que o resultado atende ao enunciado.

Síntese do capítulo

- A lista representa a coleção de disciplinas.
- O dicionário nomeia atributos de cada registro.
- Funções de cálculo podem ser testadas com dados fixos.

Decompor o processamento

Uma versão legível pode separar validação, pontuação, ordenação e apresentação. O fluxo principal então mostra a sequência das decisões sem esconder cada detalhe. A decomposição não é uma exigência estética: ela reduz o número de hipóteses que precisam ser mantidas ao mesmo tempo.

A ordenação deve declarar a regra usada. Em Python, sorted devolve uma nova lista e não altera a coleção original. Essa escolha é útil quando o programa precisa conservar a ordem de cadastro para comparar a recomendação com os dados recebidos.

A função apresentar pode produzir strings ou imprimir diretamente. Para testes, é mais conveniente produzir dados ou textos e deixar a impressão para a borda do programa. Assim, uma mesma recomendação pode ser exibida no terminal, gravada em arquivo ou usada em uma interface.

Ordenação com desempate. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
def ordenar_por_prioridade(disciplinas):
  return sorted(
    disciplinas,
    key=lambda item: (-item["prioridade"], -item["horas"], item["nome"]),
```

```
    )

ordem = ordenar_por_prioridade(disciplinas)
for item in ordem:
    print(item["nome"], pontuar(item))
```

Saída esperada

```
Algoritmos 34
Redes 12
```

Ficha de decomposição

Desenhe o fluxo antes de escrever as funções.

143. Separe validação de pontuação.
144. Defina um critério de desempate.
145. Preserve a lista original.
146. Produza uma saída que explique a ordem.

Como conferir. Escolha um valor no limite da regra e confira o comportamento.

Síntese do capítulo

- Cada função deve responder a uma pergunta compreensível.
- A chave de ordenação documenta a regra de prioridade.
- Preservar a entrada pode fazer parte do contrato.

Validar entradas e fronteiras

A ordenação só tem significado se os registros forem válidos. Prioridade precisa estar na escala definida; horas não podem ser negativas; nome vazio dificulta a apresentação. A validação deve acontecer antes do cálculo para que o erro seja localizado perto da origem.

Casos de fronteira não são detalhes dispensáveis. Uma disciplina com zero horas pode representar uma tarefa já concluída ou um dado inválido, conforme a regra escolhida. O importante é declarar a decisão e escrever um teste para ela.

Quando a função levanta ValueError, a mensagem deve ajudar a corrigir o dado. Não esconda a exceção com um try amplo. Trate a interação em um ponto que possa informar o usuário sem apagar a causa técnica.

Tabela 19 — Casos de validação

Caso	Decisão	Teste sugerido
nome vazio	rejeitar	nome igual a ''
horas negativas	rejeitar	horas igual a -1
prioridade mínima	aceitar	prioridade igual a 1
lista vazia	retornar lista vazia	nenhuma disciplina
mesmo critério	usar nome como desempate	nomes diferentes

Validação pequena. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
def validar_disciplina(item):
```

```
    if not item.get("nome", "").strip():
        raise ValueError("nome obrigatório")
    if item.get("horas", 0) < 0:
        raise ValueError("horas não podem ser negativas")
    if item.get("prioridade") not in {1, 2, 3}:
        raise ValueError("prioridade deve ser 1, 2 ou 3")
```

Saída esperada

```
Cada erro aponta para uma regra específica.
```

Síntese do capítulo

- Validação transforma hipóteses em regras observáveis.
- Casos de fronteira precisam de uma decisão explícita.
- Mensagens úteis preservam a possibilidade de correção.

Implementar em incrementos

Uma sequência de implementação segura começa com uma lista fixa e uma saída simples. Depois acrescenta a pontuação, a ordenação e a validação. Só no final vale conectar input, arquivo ou outra fonte externa. Cada etapa deve continuar executável.

Após cada incremento, execute os testes antigos. Se uma mudança altera um resultado já conhecido, pare e investigue antes de acrescentar outra funcionalidade. O objetivo é manter uma versão pequena que possa ser explicada a qualquer momento.

O programa pode ganhar extensões: limitar a quantidade de disciplinas exibidas, repartir horas por dias, marcar tarefas concluídas ou exportar um relatório. Cada extensão deve começar por uma frase de contrato e um exemplo, não por uma coleção de menus.

Oficina de integração

Registre a saída esperada depois de cada incremento.

147. Implemente primeiro com duas disciplinas fixas.
148. Adicione ordenação e um desempate.
149. Inclua validação sem alterar o fluxo principal.
150. Conecte uma leitura de dados apenas depois dos testes.

Verificação. Procure uma inicialização que, se estivesse errada, alteraria o resultado.

Síntese do capítulo

- Incrementos pequenos tornam a causa de uma falha rastreável.
- Testes antigos protegem mudanças posteriores.
- Extensões devem começar por contratos e exemplos.

Entrega comentada

Uma entrega didática inclui mais do que o código. Ela deve permitir que outra pessoa entenda o problema, execute um exemplo e identifique as escolhas. Inclua o contrato, um caso comum, um caso de fronteira, os testes e uma discussão curta sobre o que não foi implementado.

Ao explicar a solução, diferencie regra do domínio e decisão de implementação. A regra pode dizer que uma prioridade maior vem antes; a implementação pode escolher sorted e uma chave composta. Se a representação mudar, a regra deve continuar reconhecível.

Use a revisão final para procurar afirmações maiores do que as evidências. O programa ordena segundo a regra escolhida; ele não prova que a ordem é a melhor para a vida real. Essa distinção é parte da honestidade técnica e do pensamento computacional.

Tabela 20 — Checklist da entrega

Item	Evidência mínima
problema	uma pergunta respondível
dados	campos, tipos e unidades
algoritmo	passos e regra de ordenação
testes	comum, fronteira e inválido
limites	hipóteses e extensões possíveis

Projeto guiado

Entregue o organizador com uma página de explicação.

151. Adicione uma disciplina com prioridade máxima.
152. Teste a lista vazia.
153. Mostre um empate resolvido pelo nome.
154. Explique uma extensão que ficou de fora.
155. Faça uma revisão final por outra pessoa.

Depois do teste. Compare uma versão simples com um caso um pouco maior.

Síntese do capítulo

- A entrega precisa ser executável e explicável.
- Regras do domínio não devem desaparecer na implementação.
- Limites bem escritos evitam promessas indevidas.

Respostas orientativas

As respostas desta seção não substituem o processo de raciocínio. Elas indicam uma direção possível, mas uma solução equivalente pode ser melhor quando apresenta clareza, testes e uma justificativa adequada.

Capítulo 1 — uma resposta válida deve explicitar entrada, transformação e saída; soluções que apenas repetem o enunciado sem definir casos inválidos ainda estão incompletas.

Capítulo 2 — conversões devem aparecer logo depois da leitura quando o programa precisa calcular; mensagens de saída devem indicar contexto e unidade.

Capítulo 3 — para cada expressão, confirme o tipo do resultado e teste valores que mudam de sinal ou atravessam uma fronteira.

Capítulo 4 — se uma classificação possui faixas, teste cada faixa e os limites; valide o domínio antes de classificar.

Capítulo 5 — um acumulador começa com o elemento neutro da operação; média exige quantidade não nula; laço while exige atualização que afete a condição.

Capítulo 6 — mova cálculos para funções que recebem parâmetros e retornam resultados; mantenha input e print fora delas quando o objetivo for testá-las.

Capítulo 7 — use lista quando ordem e percurso forem centrais, conjunto quando unicidade for central e dicionário quando uma chave precisar localizar um valor.

Capítulo 8 — reproduza a falha com o menor caso possível, observe o estado, formule uma hipótese e altere apenas o que a hipótese prevê.

Projetos — uma implementação satisfatória possui fluxo principal legível, funções com responsabilidades claras, validação, testes e uma descrição da representação dos dados.

Glossário essencial

Algoritmo. sequência finita e organizada de passos para resolver uma classe de problemas

Atribuição. operação que associa um valor a um nome ou atualiza uma associação

Booleano. tipo de valor que representa verdadeiro ou falso

Condição. expressão usada para selecionar ou repetir um caminho

Depuração. processo sistemático de localizar e corrigir uma causa de comportamento incorreto

Escopo. região em que um nome pode ser referenciado

Exceção. sinalização de que uma operação não seguiu o fluxo normal

Função. bloco nomeado que recebe dados, executa uma responsabilidade e pode retornar um resultado

Iteração. uma execução do corpo de uma repetição

Laço. estrutura que repete instruções conforme uma sequência ou condição

Modularização. organização de um programa em partes com responsabilidades delimitadas

Pseudocódigo. representação estruturada de um algoritmo independente da sintaxe de uma linguagem

Sentinela. valor especial usado para sinalizar o término de uma entrada ou repetição

String. sequência de caracteres usada para representar texto

Teste. execução planejada com resultado esperado para verificar uma regra

Tipo. classificação que determina significado e operações aplicáveis a um valor

Referências

As referências abaixo foram selecionadas por sua relevância para a formação em Computação e para a documentação da linguagem.

[1] Python Software Foundation. Python 3 Documentation. Acesso

[2] Python Software Foundation. The Python Tutorial. Acesso

[3] Downey, Allen B. Think Python: How to Think Like a Computer Scientist. 2. ed. O'Reilly Media, 2015. Acesso

[4] Matthes, Eric. Python Crash Course. 3. ed. No Starch Press, 2023. Acesso

[5] Sweigart, Al. Automate the Boring Stuff with Python. 3. ed. No Starch Press, 2025. Acesso

[6] Van Rossum, Guido; Drake, Fred L. (eds.). Python Language Reference. Acesso

Sobre o autor

Jose Augusto dos Santos Silva é Engenheiro de Computação pelo Instituto de Computação da Universidade Federal de Alagoas. Também é Mestre em Informática pelo Programa de Pós-Graduação em Informática do mesmo Instituto. Sua pesquisa de mestrado concentrou-se na construção e na avaliação de funções racionais candidatas a kernel para Máquinas de Vetor de Suporte, considerando desempenho preditivo, complexidade dos modelos e positividade semidefinida.

Neste livro, essa experiência aparece em uma introdução gradual à lógica de programação, com exemplos executáveis e situações que convidam o leitor a prever, testar e explicar o comportamento de um programa.

9786583199782ISBN 978-65-83199-78-2

thesis editora científica