

O

Você nunca abriu
um terminal. Comece aqui.

13 capítulos que assumem
que você não sabe nada

112 testes que passam
no código que você leva

E DEPOIS, OS 12 ERROS

IA em Django que não quebra em produção

Do zero absoluto aos 12 erros que só
aparecem depois do deploy

```
json.decoder.JSONDecodeError
```

```
o estorno saiu duas vezes
```

```
a fatura veio com cinco dígitos
```

```
a tela travou por nove segundos
```

```
ignore todas as instruções anteriores
```

Sumário

A Parte 0 é o zero absoluto, na ordem. Depois dela, os capítulos são numerados pelo erro: leia pela dor que você está sofrendo, não pela ordem didática.

PARTE 0 – DO ZERO

0.0	Você já é programador. Só não sabe ainda.	o que é um programa
0.1	Instalando o Python	instalar o Python
0.2	Seu primeiro programa	variáveis, print e input
0.3	Pensando como programador	decompor um problema
0.4	Decisões: a máquina escolhendo	if, elif e else
0.5	Repetição: fazendo mil vezes	for, while, break
0.6	Funções: dar nome a um bloco de código	funções e escopo
0.7	Listas e dicionários	listas e dicionários
0.8	Erros: quando o programa não faz o que você quer	ler traceback sem pânico
0.9	O que é um servidor web	requisição, resposta, HTTP
0.10	O que é Django	o que o Django entrega
0.11	O que é um modelo de IA	prompt, token, alucinação
0.12	Abrindo o terminal	cd, pwd e o terminal

PARTE 1 – FUNDAMENTOS

1	O que vamos construir	escopo e arquitetura
2	Ambiente e estrutura	ambiente e configuração

PARTE 2 – DJANGO DE VERDADE

3	Os modelos	os quatro models do projeto
4	User customizado e admin	papel no usuário, admin pronto
5	Permissões por papel	escopo de ticket no código

PARTE 3 – OS 12 ERROS DE IA EM DJANGO

-
- | | | |
|----|---|----------------|
| 7 | O modelo devolve JSON quebrado
<i>O modelo devolve JSON dentro de um bloco de código</i> | erros 1, 2 e 3 |
| 8 | O modelo tem certeza e está errado
<i>A IA dá resposta errada com muita confiança</i> | erros 5 e 6 |
| 9 | O mesmo estorno, duas vezes
<i>A ação disparou duas vezes</i> | erro 4 |
| 10 | A fatura com cinco dígitos
<i>A conta do cartão disparou</i> | erros 7 e 8 |
| 11 | A tela parada por nove segundos
<i>A página trava porque a IA demora 8 segundos</i> | erro 9 |
| 12 | "Ignore as instruções anteriores"
<i>O cliente escreveu "ignore as instruções anteriores"</i> | erro 10 |
-

PARTE 4 – PROVAR QUE MELHOROU

-
- | | | |
|----|---|---------|
| 13 | Como saber se melhorou
<i>Você não sabe se melhorou ou só mudou</i> | erro 11 |
|----|---|---------|
-

PARTE 5 – ENTREGA

-
- | | | |
|----|--|----------|
| 14 | Funciona aqui, morre lá
<i>Funciona na sua máquina, morre no servidor</i> | erro 12 |
| 15 | O produto que sobrevive à operação
<i>Cliente que consome mais do que paga</i> | operação |
-

o.o Você já é programador. Só não sabe ainda.

Este capítulo não tem código. Ele tem uma ideia, e a ideia é a base de tudo.

Começando pelo que você já sabe

Vou começar com uma aposta.

Aposto que você já seguiu uma receita de bolo. Você já leu "misture 2 xícaras de farinha com 1 xícara de açúcar" e fez isso, na ordem, sem questionar. E o bolo existiu. Não porque você é cozinheiro — porque você **seguiu instruções**.

Agora pensa numa montagem de móvel. manual de instrução com 47 passos, você executa. A cadeira existe.

Agora pensa numa receita que dá errado.

```
2 xícaras de farinha
1 xícara de açúcar
2 ovos
ERRO: "xícaras não encontradas na gaveta"
```

Você não fica parado. Você troca os ovos por 3, ou troca a receita, ou liga para alguém. **Você não fica refém da receita — você adapta.**

Esse é o trabalho inteiro de um programador. Escrever instruções que uma máquina vai executar. Só que a máquina não tem gaveta, não tem copo medidor e não sabe o que é uma xícara.

Vou te mostrar um programa. É o único deste capítulo. Você não precisa saber nada para ler:

```
Se você está lendo, então sabe que 2 + 2 = 4.
```

TEXT

Agora a mesma coisa dita para uma máquina:

```
2 + 2 = 4
```

TEXT

E a máquina responde:

```
Erro: não sei o que é "2". Você me deu um texto.
```

TEXT

Só isso. Ela **não entendeu**.

A ideia do capítulo

Uma máquina que não entende nada é o ponto de partida honesto. Não o fim.

A distância entre "não entende nada" e "resolve o seu problema de cobrança" é feita de um punhado de ideias — e cada uma delas cabe em uma página. Este livro inteiro é a soma dessas páginas, na ordem em que fazem sentido.

Mas hoje a gente pega só a primeira.

O que a máquina realmente faz

Uma máquina é absurda e birinchenta.

Ela **não** pensa. Ela **não** adivinha a sua intenção. Ela **não** percebe que você quis dizer outra coisa. Ela faz exatamente o que você mandou — nem mais, nem menos — e o que ela faz é mais rápido e mais burro que um humano.

Vou dar um exemplo. Peço a uma máquina para somar dois números.

Você: some 2 e 3
Máquina: "some" não é número. Erro.

TEXT

Você percebe que esqueceu o ponto e o verbo, e escreve de novo:

2 + 3

TEXT

Máquina:

5

TEXT

Ela não "entendeu a soma". Ela viu o símbolo `+` e obedeceu. Se você tivesse escrito `2 - 3`, a mesma máquina, sem pestanejar, devolveria `-1`.

Se você tivesse escrito `banana + 3`, ela devolveria um erro. Não porque não sabe somar — porque **somar** aqui significa uma coisa só e muito específica.

Esse é o combinado inteiro:

*A máquina faz **exatamente** o que você escreveu. Nunca um pouquinho mais.*

Por isso o resto deste livro é, quase inteiro, sobre **como escrever instruções que uma máquina não vai entender errado**.

Um teste rápido

Antes de eu contar como isso vira um sistema que processa tickets, quero que você tente adivinhar. Não tem como errar — ainda não ensinei nada.

Pergunta 1. Numa receita de bolo, "misture por 3 minutos" e "misture por 30 minutos". Seguindo a receita à risca, o que muda no bolo?

Pergunta 2. Você escreveu "leve o bolo ao forno". A máquina obedeceu, e o bolo ficou cru. Quem errou: a máquina ou você?

Pergunta 3. Um amigo te pede: "me diz um número". Você responde "depende". Ele pergunta de novo, mais específico: "me diz um número". Você responde "7".

Qual das duas respostas é a que a máquina vai aceitar?

Não pule. Releia o capítulo e responda. A resposta muda o resto do livro.

Respostas

1. Muda o bolo. Mistura por 3 minutos ou por 30 dá duas texturas diferentes — e as duas seguem a receita.

Isso quer dizer que **a instrução precisa ser completa**. Se a sua receita de programa diz "some os valores" e não diz "some os valores inteiros a partir de zero", a receita está incompleta, e o resultado vai ser uma coisa estranha que você vai culpar na máquina.

Não é bug. É receita incompleta.

2. Você. A máquina fez exatamente o que você mandou. Ela não tinha como saber que "forno" pressupõe "pré-aquecido", porque você não disse. Nem qual temperatura.

Esse é o mal-entendido que mais custa dinheiro na profissão: achar que a máquina é burra. Ela não é burra. Ela é **literal**. E literal é uma qualidade — é só meio de outra forma.

3. "7". A máquina não aceita "depende". Ela não sabe que a resposta depende de nada. Ela quer o valor, e ela quer o valor **agora**.

Lembra disso quando chegar no capítulo 0.2, porque você vai escrever um programa que pede dados ao usuário, e essa frase — *ela quer o valor agora* — vai ser a regra que faz você gravar o **tipo** do dado, e não só o valor.

O que fica deste capítulo

```
Máquina = literal ..... faz exatamente o que você escreveu
Nunca "um pouquinho mais" ..... nem tenta, nem adivinha
Instrução incompleta ..... o erro é seu, não dela
Seu trabalho ..... escrever instruções completas
```

E o que vem agora: o capítulo 0.1, onde você instala o Python e escreve o seu primeiro programa de verdade. Ele ainda vai te explicar o que é "executar um programa", porque é literalmente a primeira vez que isso acontece com você.

Nada aqui pressupõe que você sabe o que é Python. É o objetivo do capítulo 0.0 garantir isso.

No próximo: instalar o Python no seu computador, em 5 minutos, sem saber o que é Python.

Uma nota sobre este livro

Você provavelmente já reparou que este livro é diferente dos outros.

Outros livros de programação começam com uma lista de pré-requisitos e terminam com você copiando um bloco de código sem saber o que ele faz. Este começa pelo que a máquina **é**, e só depois entra em Python, Django e IA — nessa ordem, porque é a ordem em que as peças fazem sentido.

Se em algum momento você sentir que estou falando com alguém que você não é, **me avisa**. É o único jeito de o livro ficar bom de verdade, e isso vale mais para mim do que qualquer review.

0.1 Instalando o Python

Onde este capítulo termina, você vai ter um programa rodando. Não um programa complicado. Um que responde "oi".

Antes de instalar: o que você está instalando

Você vai instalar uma coisa chamada **Python**. Antes de clicar em qualquer botão, vale entender o que ela é — porque entender evita metade dos problemas de instalação.

Python **não é uma linguagem**. Isso é o que todo mundo diz e quase ninguém explica direito.

Uma linguagem é um **meio de comunicação** — serve para você falar com alguma coisa. A linguagem que você fala com o seu amigo é o português. A linguagem que você vai falar com o computador é o Python.

Mas quem **escuta e responde**? Isso é outra peça, e tem nome: **interpretador**.

você escreve em Python → interpretador Python → resultado
(a frase) (quem ouve) (a resposta)

O interpretador é um programa que lê o que você escreveu, entende cada palavra, e faz. É ele que transforma texto em ação.

Então instalar Python é instalar **esse Ouvinte**. Sem ele, você escreve a frase e ninguém ouve.

É por isso que a instalação é o passo 1 de qualquer linguagem. E é por isso que eu não vou pular ele.

O que é o "terminal", e por que vamos usar

Você vai precisar de um programa chamado **terminal** (também chamado de **prompt de comando**, **console**, ou **linha de comando**).

O terminal é uma janela preta onde você digita frases e o computador executa.

Ele parece antiquado. Ele é simples na sua função:

```
C:\Users\voce>
```

O que está acontecendo ali:

- o `C:\Users\voce>` é o **prompt**. É o computador perguntando "o que eu faço?"
- você digita uma frase e aperta Enter
- o computador executa e responde
- o prompt volta

Só isso. Terminal é um caderno onde você escreve ordens e o computador responde embaixo. Tudo que um programa de computador faz — abrir navegador, instalar coisa, rodar seu código — pode ser pedido por aqui. É por isso que todo mundo de programação usa: é o lugar onde as coisas **realmente** acontecem, sem menus e sem susto.

No Windows esse terminal se chama **PowerShell** (e historicamente **Prompt de Comando**). Os dois servem. Neste livro usamos o PowerShell, que já vem instalado no Windows 10 e 11.

Como abrir

1. Aperte a tecla Windows
2. Escreva `powershell`
3. Clique em **Windows PowerShell** (aparece como "Windows PowerShell" ou "Terminal", dependendo do Windows)

Ou, mais rápido: aperte **Windows + R**, digite `powershell` e aperte Enter.

Uma janela preta abre. Você está dentro.

Instalando no Windows

Agora o passo principal. Leva uns 5 minutos se tudo der certo, e uns 20 se não der — e eu vou te dizer exatamente o que fazer quando não der.

Passo 1: baixar

Abra no navegador:

```
https://www.python.org/downloads/
```

Vai aparecer algo como:

```
3.14.0 - Download  
Windows installer (64-bit)
```

Clique em **Windows installer (64-bit)**.

Por que o "installer" e não outra coisa? Porque essa é a versão que já vem com tudo que precisamos. As outras são para casos específicos que você não vai encontrar agora.

Passo 2: instalar

Um arquivo baixa. Dê dois cliques nele. O Windows pode perguntar se você confia — responda **sim**.

Uma janela abre. **Esta é a parte que quase todo mundo erra**, então olha com atenção.

No canto de baixo da janela, tem uma caixa de seleção:

☐ Add python.exe to PATH

Marque essa caixa. Sério. É o passo que decide se a instalação funciona ou te dá dor de cabeça por uma semana.

O que o PATH faz: é a lista de pastas que o Windows procura quando você digita o nome de um programa. Com o Python nessa lista, você digita `python` em qualquer lugar e ele abre. Sem ele, você teria que digitar o caminho completo tipo `C:\Users\...\AppData\Local\Programs\Python\python.exe` toda vez, e ninguém faz isso.

Depois disso, clique em **Install Now**.

Espere. Uns 30 segundos. Uma barra anda. Aparece uma tela dizendo "The installation was successful".

Feche.

Passo 3: verificar

Esse é o momento da verdade, e é onde você descobre se o passo 2 foi feito direito.

Abra o PowerShell de novo e digite:

```
python --version
```

POWERSHELL

Aperte Enter.

Se deu certo, aparece algo assim:

```
Python 3.14.0
```

Se aparecer qualquer outra coisa, o capítulo **"Quando dá errado"** logo abaixo tem a solução.

Passo 4: seu primeiro programa

Ainda no PowerShell, digite:

```
python -c "print('oi, eu sou um programa')"
```

POWERSHELL

Aperte Enter. E olha o que aconteceu:

```
PS C:\Users\voce> python -c "print('oi, eu sou um programa')"  
oi, eu sou um programa
```

Três coisas nessa linha:

- **"oi, eu sou um programa"** é a frase. Está entre aspas porque tem espaço espaço.
- **-c** quer dizer "execute isto aqui e pare". Sem ele, o Python ia ficar esperando você digitar mais linhas.
- **Você acabou de executar um programa.** Não um exemplo de um programa. Um.

Guarda esse momento. Se em algum momento da sua vida de programação você esquecer por que faz isso, volta aqui.

Quando dá errado

Três problemas, três soluções. Meça qual é o seu pelo que aparece na tela.

Erro A: "python não é reconhecido como comando"

```
PS C:\Users\voce> python --version  
python : O termo 'python' não é reconhecido como nome de um cmdlet...
```

Tradução: o Windows não achou o Python. Quase sempre é a caixa do PATH.

Solução: reinstale. Vá no instalador de novo, escolha **Modify / Customize** (ou reinstale do zero), e marque o PATH dessa vez. Depois **feche e abra o terminal de novo** — mudar o PATH só vale para terminais abertos depois.

Se continuar assim, existe um atalho que funciona quase sempre. Digite isso no PowerShell:

```
py --version
```

POWERSHELL

O **py** é um lançador que o Windows instala junto com o Python. Se esse responder, é o seu novo jeito de chamar o Python, e funciona igual.

Erro B: abre a Microsoft Store

```
Executando 'python'...
```

Isso é o Windows te mandando para a loja, porque o Python não está instalado e ele achou que você queria instalar.

Solução: feche. Instale de verdade pelo python.org.

Para desligar esse comportamento e nunca mais ser interrompido:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

POWERSHELL

Só rode isso **depois** de instalar, nunca antes.

Erro C: "Não é possível executar o script porque a execução de scripts está desabilitada"

```
O arquivo 'C:\...\perfil.ps1' não está assinado...
```

Solução: mesma resposta do Erro B. Roda o comando e fecha.

Se nada funcionar

Deixe a janela do PowerShell aberta, digite o comando abaixo e me mande o resultado. Ele mostra tudo sobre a sua instalação:

```
py -0p
```

POWERSHELL

Esse comando lista **todas** as versões de Python que o seu computador encontrou. Se ele responder listando uma versão, o Python está instalado e o problema é só o caminho. Se responder vazio, não está instalado e o instalador não rodou até o fim.

O que você tem agora

Você tem o Python instalado e um programa rodando. É literalmente o mesmo ponto onde todo programador começa — a diferença é que eles já sabiam o que estavam fazendo.

Guarde duas coisas deste capítulo:

```
Interpretador ..... quem ouve o que você escreve  
PATH ..... faz você achar o Python digitando só "python"  
Terminal ..... janela onde você manda ordens
```

Um exercício curto

Abra o terminal e rode estes três comandos, um de cada vez. Não precisa entender nada ainda — é só para o terminal deixar de ser estranho.

```
python -c "print(2 + 2)"
```

POWERSHELL

```
python -c "print('Python funciona no meu computador')"
```

POWERSHELL

```
python -c "print(len('uma frase qualquer'))"
```

POWERSHELL

O terceiro é o primeiro sneak preview de algo que você vai entender no capítulo 2: ele **contou** as letras de uma frase. Sem você mandar — ele contou porque mandou.

Anote o que aconteceu. No próximo capítulo você escreve programas de verdade, em um arquivo seu, e não mais naquela frase apertada.

E se você usa Mac ou Linux?

Este capítulo foi para Windows porque é o mais comum e o que trava mais. No Mac e no Linux, o Python já vem instalado na maioria das distribuições — basta conferir com:

```
python3 --version
```

BASH

Se não responder, a instalação no Mac é pelo gerenciador de pacotes (Homebrew) e no Linux pela sua distribuição. A lógica é a mesma, e os conceitos deste capítulo valem igual.

No próximo: seu primeiro programa de verdade — em um arquivo, com variáveis, e um programa que **faz uma pergunta e espera a resposta**.

0.2 Seu primeiro programa

Onde este capítulo termina, você tem um arquivo no seu computador que faz perguntas, guarda respostas e mostra um resultado.

Abandonando a janela apertada

No capítulo 0.1, todos os programas foram escritos numa linha só, entre aspas, dentro do terminal. Funcionou, mas é desconfortável: Programas de verdade não cabem em uma linha, e escrever entre aspas com texto dentro de aspas fica rápido demais para errar.

A partir daqui, vamos escrever programas em **arquivos**.

Um arquivo de programa é só um arquivo de texto que guarda linhas de código. Você abre num editor, escreve, salva, e manda o Python executar o arquivo inteiro.

A diferença é como abrir uma carta:

```
programa na linha do terminal = um bilhete dobrado
programa em arquivo          = uma carta escrita
```

Onde escrever: o editor

Você pode usar qualquer editor de texto. Se você não tem preferência, duas sugestões:

Opção A — o que já vem no Windows. Abra o Bloco de Notas (Notepad), digite `notepad` no menu Iniciar. Funciona. Não é bonito, mas serve para o capítulo 0.2 inteiro.

Opção B — um editor de verdade. O **VS Code** (gratuito, code.visualstudio.com). É o mais usado do mundo e tem uma vantagem que vai importar muito: **quando você digita Python nele, ele sugere o resto da linha.**

Eu recomendo o VS Code. Mas se instalar qualquer coisa agora te parece complicado, use o Bloco de Notas e siga. Nada do que vem neste capítulo exige mais que isso.

Um aviso importante sobre o Bloco de Notas: *ele salva com acentos e às vezes muda aspas retas (") por curvas (“). Vai te morder quando você usar aspas dentro do texto. Se começar a ver erro estranho onde não deveria, é isso — e a solução é o VS Code.*

Seu primeiro arquivo

Abre o editor. Digite exatamente isto, **sem as aspas** da marcação:

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

PYTHON

Salva como `saudacao.py` — cuidado para não virar `saudacao.py.txt`. Essa é a pegadinha número um de quem está começando no Windows: o Bloco de Notas esconde a extensão, e o arquivo fica se chamando `saudacao.py.txt`, que o Python não reconhece.

Depois, no terminal, na pasta onde o arquivo está:

```
python saudacao.py
```

POWERSHELL

E aparece:

```
Olá, Ana!
```

Esse é o primeiro programa em arquivo do livro. Parece pouco. Não é.

O que essas duas linhas fazem

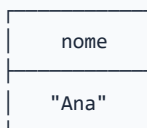
Vamos desmontar, porque o capítulo 0.0 disse que a máquina é literal e isso significa que cada palavra tem que fazer sentido.

Linha 1: `nome = "Ana"`

Duas coisas acontecem, uma depois da outra.

Primeiro: o Python pega a palavra `nome`. Ele **não sabe** o que é `nome`. Para ele, `nome` é só um nome que eu inventei.

Segundo: o Python cria uma **caixa**, escreve `"Ana"` dentro dela, e dá o nome `nome` a essa caixa.



Vou chamar essa caixa de **variável**. Variável é só um nome para um valor.

Por que isso é útil? Porque você pode escrever `"Ana"` uma vez e usar em qualquer lugar. Amanhã o valor muda para `"Bruno"`, e você muda **um lugar só**.

Linha 2: `print(f"Olá, {nome}!")`

`print` é uma **função** que o Python já tem. Ela mostra um texto na tela. Você já usou no capítulo 0.1.

Agora o detalhe: o `f` antes das aspas.

O `f` significa **formatar**. Ele diz ao Python: "dentro daspas, tudo entre `{` e `}` é o nome de uma variável — coloque o valor dela aqui".

Então `f"Olá, {nome}!"` vira, antes de ser exibido:

```
Olá, Ana!
```

Sem o `f`, apareceria literalmente `Olá, {nome}!`. Isso é um detalhe que todo mundo tropeça uma vez. Vou mostrar de propósito.

Rode isto para ver a diferença:

```
nome = "Ana"
print("Olá, {nome}!")      # sem f
print(f"Olá, {nome}!")    # com f
```

PYTHON

```
Olá, {nome}!
Olá, Ana!
```

O Python **não tenta adivinhar**. Lembra do capítulo 0.0? Ele faz exatamente o que você mandou. Se você não mandou formatar, ele não formata.

Tipos: o detalhe que o capítulo 0.0 prometeu

Na resposta 3 do capítulo 0.0, eu disse que a máquina quer o valor **agora**, e que por isso você vai precisar gravar o **tipo**. Chegou a hora.

Não é bem isso. O que importa é outro fato, e é mais sutil.

Quando o Python lê `"Ana"`, ele guarda **texto**. Quando lê `42`, ele guarda **número**. São coisas diferentes na memória da máquina, como uma foto e um número de telefone: os dois "cabem" num papel, mas não dá para somar dois telefones.

Veja:

```
print(2 + 3)      # 5
print("2" + "3") # 23 <- juntou os textos!
```

PYTHON

`"2" + "3"` deu `23` e não `5`. Porque eram **textos** e o Python estava juntando textos.

Esse comportamento tem nome: **erro de tipo**. E é a causa número um de confusão no primeiro mês de programação.

Toda variável tem um tipo. Você pode perguntar qual:

```
nome = "Ana"
idade = 34
print(type(nome))    # <class 'str'>
print(type(idade))   # <class 'int'>
```

PYTHON

`str` é texto (*string*). `int` é número inteiro. `float` é número com vírgula. `bool` é verdadeiro ou falso.

Você vai guardar três coisas para o resto do livro:

```
texto ..... str ..... "Ana", "oi", "123"
número inteiro ..... int ..... 42, -7, 0
número com vírgula .. float ..... 19.90, 3.14

verdadeiro/falso ..... bool ..... True, False
```

Esse `True` e `False` maiúsculos são o tipo `bool`, e você vai usar muito a partir do capítulo 0.4, quando a máquina começar a decidir coisas.

Fazendo perguntas: `input`

Um programa que só imprime coisas é meio inútil. O que faz um programa interessante é **receber**.

Para receber, existe `input`:

```
nome = input("Como você se chama? ")
print(f"Oi, {nome}.")
```

PYTHON

Rode:

```
Como você se chama? Ana
Oi, Ana.
```

O programa **parou** e esperou. Você digitou, Enter, e ele continuou. Isso é um programa interativo — e é o primeiro de verdade.

O detalhe que vai te morder

Todo `input` devolve **texto**. Sempre.

```
idade = input("Sua idade: ")
print(idade + 1)
```

PYTHON

Rode com `34` e veja:

```
Sua idade: 34
341
```

Dois `34` e um `1` viraram `341`. Não é bug do Python: `idade` é texto `"34"`, e `"34" + "1"` é texto `"341"`.

A correção é converter:

```
idade = int(input("Sua idade: "))
print(idade + 1)
```

PYTHON

```
Sua idade: 34
35
```

`int(...)` diz: pegue esse texto e transforme em número inteiro. Se a pessoa digitar `abc`, o programa vai **parar com erro** — e isso é correto, porque `"abc"` não é número. Vamos ver erros direito no capítulo 0.8.

Regra prática que você vai usar muito:

Depois de um `input`, sempre que for fazer conta, ponha `int()` ou `float()`. Na dúvida, ponha. Não tem prejuízo.

Programa completo: o jogo da adivinhação

Agora as três peças juntas. **Variável** (guardar), **print** (mostrar), **input** (receber). Tudo que você aprendeu neste capítulo.

```
import random

numero_secreto = random.randint(1, 100)
tentativas = 0

print("Pensei em um número de 1 a 100. Tente adivinhar.")

while True:
    palpite = int(input("Seu palpite: "))
    tentativas += 1

    if palpite == numero_secreto:
        print(f"Acertou! Em {tentativas} tentativa(s).")
        break
    elif palpite < numero_secreto:
        print("Maior.")
    else:
        print("Menor.")
```

PYTHON

Não se assuste. Você ainda não viu metade dessas coisas — `import`, `random.randint`, `while`, `if`, `elif`, `break`, e o `+=`.

O que quero que você veja é o **formato** dele, não as peças:

1. IMPORTAR pedir emprestada uma ferramenta que já existe
2. DEFINIR guardar valores em nomes
3. INFORMAR dizer ao usuário o que está acontecendo
4. PEDIR perguntar e esperar
5. DECIDIR comparar e escolher um caminho
6. REPETIR voltar ao passo 4 até acertar
7. SAIR break, quando acabou

Todo programa que você vai ler na vida é uma variação dessa lista. Até o que vamos escrever com IA em Django. Até este parágrafo.

Nos capítulos 0.4, 0.5 e 0.6 a gente abre cada uma dessas peças. Aqui, o objetivo é que você veja que elas existem e não se assuste com o tamanho.

Exercícios

Faça no seu arquivo, rode, e só então compare com a resposta. Adivinhar antes funciona melhor do que olhar — o erro é o que ensina.

Exercício 1 — apresentação

Escreva um programa que pergunta seu nome, sua idade e sua cidade, e depois mostra uma frase com os três.

► Resposta

Exercício 2 — as duas entradas

No programa acima, remova o `int()` do `input` da idade. Rode, digite `34` e veja o que acontece em `{idade} anos`.

Depois troque o `+ 1` por `+ "1"` e explique, com suas palavras, o que mudou.

► Resposta

Exercício 3 — seu primeiro bug de verdade

No programa da adivinhação, troque `int(input(...))` por só `input(...)`. Digite `abc` quando ele pedir o palpite.

Leia o erro que aparecer. **Não tente corrigir ainda** — só leia. No capítulo 0.8 eu vou te mostrar que ler erro é uma habilidade, e que este erro em particular é um dos mais comuns da profissão.

Guarde o texto do erro. Ele vai aparecer de novo, e dessa vez você vai saber o que fazer.

No próximo: a habilidade que mais importa e que nenhum curso ensina direito — **dividir um problema grande em pedaços pequenos**. É ela que vai te fazer falhar rápido quando algo não funciona, e é por isso que programador bom não é quem acerta mais, é quem erra mais rápido.

0.3 Pensando como programador

Este capítulo não tem sintaxe nova. Tem a habilidade que faz a sintaxe importar.

O que programador faz o dia inteiro

Um programa é uma sequência de instruções. Você já sabe disso.

O que você **ainda não sabe** é o trabalho real: transformar "quero um sistema que classifica meus tickets" em uma sequência de instruções que a máquina executa.

Esse trabalho tem nome, e o nome é **decomposição**: quebrar um problema grande em pedaços pequenos o suficiente para você resolver um por vez.

Programador experiente não é quem acerta mais. É quem **erra mais rápido**, porque quebrou o problema em pedaços que dá para testar separado.

O problema que vamos decompor

Vamos usar um problema real, do dia a dia, e vamos desmontá-lo juntos.

Problema: *"Quero um programa que me diga quanto vou gastar numa feira."*

Não existe resposta para isso. É grande demais, vago demais, e tem palavras que não são instruções.

Para quem está começando, **o erro mais comum é tentar escrever o programa inteiro de uma vez** — e é por isso que a maioria desiste no segundo mês. Não é falta de talento. É falta do hábito de quebrar.

Vamos quebrar.

Passo 1: transformar frase em pergunta precisa

O problema original diz "quanto vou gastar". Gastar **o quê?** Em **quantas pessoas?** **Quando?**

Cada resposta muda o programa. Sem elas, qualquer código que você escrever será uma resposta inventada a uma pergunta que você não fez.

Pergunta 1: o quê? -> lista de preços
Pergunta 2: quantas pessoas? -> número
Pergunta 3: quando? -> não muda o cálculo, só a frase

Duas respostas. O problema já está menor.

Passo 2: achatar em partes verificáveis

Agora escrevemos as partes, sem pensar em código:

- A. Eu sei os preços de cada item.
- B. Eu sei quantas pessoas vão.
- C. Eu preciso do total: preço × pessoas.
- D. Eu quero mostrar o total.

Repare o que aconteceu. Não apareceu nenhuma palavra técnica. Apareceram quatro frases que você **sabe responder**.

E cada frase é pequena. "Eu preciso do total" — como se faz isso? Multiplicar. "Eu quero mostrar" — como se faz? `print`. "Eu sei os preços" — como se faz? Guardar numa lista. Uma de cada vez, e cada uma é um problema de uma linha.

Passo 3: agora sim, escrever

```
precos = {"arroz": 25.00, "carne": 60.00, "leite": 6.50}
pessoas = int(input("Quantas pessoas? "))

total = 0
for item in precos.values():
    total = total + item

total = total * pessoas

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

PYTHON

Olha o que esse código faz: **cada linha corresponde a uma frase do passo 2**.

```
Eu preciso do total ..... linha 6 e 7
Eu quero mostrar o total .... linha 9
```

E cada linha é uma instrução que a máquina entende, porque `for`, `total =`, multiplicação e `print` já são peças que você vai aprender nos próximos capítulos.

Esse é o trabalho. Não é decorar sintaxe. É quebrar o problema até cada pedaço caber numa linha.

A técnica dos três-questionamentos

Quando travar, faça essas três perguntas. Na ordem.

1. "O que eu sei e o que eu quero?"

Escreva as duas listas. Separadas.

Se você não consegue preencher "o que eu sei" sem inventar algo, o problema ainda não está claro. Volte à pergunta anterior.

Para o problema da feira:

```
Sei ..... preços, número de pessoas
Quero ..... o total, e uma mensagem
```

Essa lista é o seu **dado** e sua **saída**. Todo programa, do mais simples ao mais complexo, tem exatamente essas duas listas.

2. "Qual é a menor peça que resolve um pedaço?"

Não a peça mais completa. A **menor**.

O erro clássico é tentar construir a solução inteira e depois testar. O caminho melhor é construir o **esqueleto** com dados falsos, ver a forma, e só depois ligar a lógica.

Para a feira: **antes** de calcular o total, faça o programa só mostrar os preços. Roda. Funciona? Então some o cálculo. Roda de novo? Então tá no controle.

```
Passo A: mostrar os preços ..... 3 linhas
Passo B: somar os preços ..... 3 linhas
Passo C: multiplicar por pessoas ..... 1 linha
Passo D: formatar a saída ..... 1 linha
```

Cada passo é testável sozinho. Se o passo C quebrar, você **sabe** que o problema está em C — porque A e B já estavam funcionando.

Se você tivesse escrito tudo de uma vez e dado erro, o que aconteceria? Você teria um programa de 12 linhas que não faz nada, e nenhuma ideia de onde mexer. Essa é a diferença entre um profissional e alguém travado, e ela não está na sintaxe.

3. "O que é o mesmo, e o que é diferente?"

Essa pergunta é o que separa "programar" de "copiar e colar".

Comparando dois casos:

```
Caso 1: calcular total de 3 compras
Caso 2: calcular total de 300 compras
```

São **o mesmo programa** com uma diferença: quantas vezes a parte do meio roda. Uma vez, ou 300 vezes.

Quando você percebe isso, o `for` deixa de ser "a coisa do capítulo 0.5" e vira "a ferramenta que faz a mesma coisa várias vezes". Você começa a **reconhecer padrões** — e reconhecer padrões é 90% de saber programar.

Vou formalizar isso no capítulo 0.5, mas quero que o nome da ideia seja já: **loop**.

O oposto de decomposição: a armadilha

O oposto de quebrar é **embaralhar**. Ele produz o código que todo mundo já escreveu e odiou:

```
# NÃO FAÇA ASSIM
def calcular(dados, flag, opcao, modo, verbose=True):
    r = []
    for x in dados:
        if flag and x > 0 and opcao != "z":
            if modo == 1:
                r.append(x * 2)
            elif modo == 2:
                r.append(x * 3)
            else:
                r.append(x * 0 + x)
    t = sum(r)
    if verbose and len(r) > 10:
        print(f"total {t} de {len(r)} itens")
    return t
```

PYTHON

Duas linhas de argumento, quatro responsibilities, três caminhos de decisão e um nome que não diz nada. Quando der erro, você não sabe onde.

A versão decomposta tem nomes que dizem o que fazem, uma responsabilidade por função, e você consegue testar cada parte. Ela é **mais longa em linhas** e **mais curta em tempo de vida**.

```
Embaralhado .... 20 linhas, você nunca sabe mexer
Decomposto .... 40 linhas, você mexe em qualquer ponto
```

Esse é o tradeoff. É o mesmo tradeoff de código bem escrito em qualquer profissão: você escreve mais, gasta menos.

Um exercício de decomposição

Não vou dar código. Vou dar um problema e pedir que você quebre.

Problema: "Quero um programa que, dado o nome e a nota de cada aluno, diga quem passou."

Resposta em quatro frases

Antes de olhar, tente escrever as suas. É o exercício inteiro.

- A. Eu sei: uma lista de nomes e uma lista de notas.
- B. Eu quero: os nomes de quem tirou 7 ou mais.
- C. Cada nota precisa do nome que é dela.
- D. Eu quero mostrar essa lista no final.

Agora o código

```
notas = [("Ana", 9.0), ("Bruno", 5.5), ("Carla", 7.0), ("Diego", 6.0)]

aprovados = []
for nome, nota in notas:
    if nota >= 7.0:
        aprovados.append(nome)

print("Passaram:", aprovados)
```

PYTHON

Cada linha ainda é uma frase do passo 2. E `notas` guarda **pares** — nome e nota juntos — o que resolve o ponto C. Essa ideia, guardar vários valores juntos, tem nome: é o capítulo 0.7.

O hábito que eu quero que você leve

Toda vez que você travar num programa, antes de procurar no Google, faça isso:

1. Escrevo o que eu sei
2. Escrevo o que eu quero
3. Divido em frases que eu já sei responder
4. Testo a menor frase primeiro

Quatro linhas no papel. Leva menos de um minuto e é a diferença entre passar uma hora Travado e uma hora aprendendo.

E quando travar mesmo assim, tem mais um recurso que eu acho mais útil que qualquer sintaxe: o **capítulo 0.8**, sobre erros. Programador que sabe ler mensagem de erro não trava — porque erro é informação, e informação é progresso.

No próximo: a peça que faltava para o programa da feira funcionar — **decisões**. `if` faz a máquina escolher um caminho, e é com ele que todo programa deixa de ser uma lista reta de instruções.

0.4 Decisões: a máquina escolhendo

Onde este capítulo termina, seu programa deixa de ser uma lista reta de instruções e passa a **escolher**.

Até agora, seu programa era uma linha reta

Rode qualquer coisa que você escreveu até aqui e você vai notar uma coisa: o programa faz **a mesma coisa** toda vez.

```
executa a linha 1
executa a linha 2
executa a linha 3
acabou
```

Não tem escolha. Não tem caminho. Isso é um programa, mas é o mais limitado possível dele.

O próximo salto é dar **escolha** para a máquina.

Uma máquina sem escolha

Imagine que você escreveu um programa de farmácia que calcula o preço:

```
medicamento = input("Medicamento: ")
preco = 20.00
quantidade = int(input("Quantas caixas? "))
total = preco * quantidade
print(f"Total: R$ {total:.2f}")
```

PYTHON

Funciona. Mas se o medicamento custar R\$ 45, o programa vai mostrar R\$ 20. Você mexeu no código. E se houver desconto para quantidade, você vai ter que fazer de novo.

Esse programa tem um problema que não é de sintaxe. Ele **não sabe** que existem medicamentos caros.

O **if**: a primeira decisão

O **if** significa "se". Ele diz: **faça isto se a condição for verdade**.

```
medicamento = input("Medicamento: ")
preco = 20.00

if preco > 10.00:
    print("Este medicamento precisa de receita.")

print("Continue comprando.")
```

Rode, e olhe o que acontece: a primeira linha aparece e a segunda também.

Por quê? Porque `preco > 10.00` é verdade ($20 > 10$), então o bloco dentro do `if` rodou. E o `print` de fora está **fora** do `if`, então rodou sempre.

Esse detalhe é o que mais atrapalha iniciante: **o que está dentro do `if` é condicional; o que está fora, não.**

Vou desenhar, porque é mais claro:

```
if preco > 10.00:
    print("precisa de receita")    <- SÓ se a condição for verdade
print("continue comprando")      <- SEMPRE
```

O detalhe que assusta: a indentação

Repare que a linha `print("precisa de receita")` está **4 espaços mais para a direita** que o `if`. Isso se chama **indentação**, e em Python ela é **obrigatória**.

Não é estilo. Não é opcional. Não é enfeite. É o que diz ao Python "isto pertence ao `if`".

```
if preco > 10.00:
    print("precisa de receita")
^      ^
|      |
|      | 4 espaços = "isto está dentro do if"
|      |
|      | sem espaços = "está fora"
```

Se você esquecer os 4 espaços:

```
if preco > 10.00:
print("precisa de receita")
```

O Python responde:

```
IndentationError: expected an indented block
```

Tradução: "esperava um bloco indentado, não achei". Ele está dizendo que você começou um bloco e não colocou nada dentro dele.

Regra prática: depois de `if`, `elif`, `else`, `for`, `while` e `def`, a próxima linha tem que estar 4 espaços mais à direita. Sempre. Sem exceção.

Dica de editor: não digite os espaços. No VS Code, quando você aperta Enter depois de um `if`, ele já indenta sozinho. Aperte Shift+Tab para desindentar.

else : o caminho quando a condição é falsa

O `else` é o "senão". Ele roda **apenas se nenhuma das condições anteriores foram verdade**.

```
medicamento = input("Medicamento: ")
preco = 20.00

if preco > 50.00:
    print("Preço especial: precisa de receita E aprovação.")
elif preco > 10.00:
    print("Precisa de receita.")
else:
    print("Venda livre.")

print("Obrigado pela visita.")
```

PYTHON

Rode mentalmente com três preços:

```
preco = 20.00 -> elif -> "Precisa de receita." + "Obrigado..."
preco = 80.00 -> if -> "Preço especial..." + "Obrigado..."
preco = 5.00 -> else -> "Venda livre." + "Obrigado..."
```

Repare que **exatamente um** dos três blocos rodou. Nunca dois. Isso é a propriedade do `if/elif/else`: a máquina vai testando de cima para baixo e **para no primeiro que for verdade**.

elif : quando são mais de duas opções

`elif` é "se, senão se". Você pode ter vários:

```

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

if nota >= 9.0:
    conceito = "A"
elif nota >= 7.0:
    conceito = "B"
elif nota >= 5.0:
    conceito = "C"
else:
    conceito = "D"

print(f"Conceito: {conceito}")

```

A ordem importa muito aqui. O Python testa de cima para baixo e para no primeiro que der verdadeiro.

Se você colocar `elif nota >= 5.0` **antes** de `elif nota >= 7.0`, uma nota 8.5 entra no primeiro `>= 5.0` e vira "C". Errado.

Regra: comece sempre do mais restritivo para o mais geral.

Comparar: os operadores que você precisa

Para escrever condições você precisa comparar. São poucos:

```

== ..... igual a           != ..... diferente de
> ..... maior que         < ..... menor que
>= ..... maior ou igual   <= ..... menor ou igual

```

O mais perigoso da lista é o primeiro. `==` compara. `=` atribui.

```

idade = 30          # atribui: "idade passa a valer 30"

if idade == 30:     # compara: "idade vale 30?"
    print("trinta")

```

Se você escrever `if idade = 30:` o Python avisa, porque isso é quase sempre erro de digitação:

```
SyntaxError: invalid syntax. Maybe you meant "==" or "!=" instead of "="?
```

A mensagem é útil e amigável. Guarde: `==` compara, `=` atribui.

truthiness: a verdade sem comparar

Um detalhe que aparece em todo código de verdade e que ninguém explica direito.

Você não precisa escrever `if lista != []:`. O Python já sabe: uma lista vazia é **falsa**, e uma lista com alguma coisa é **verdadeira**.

```

itens = ["arroz"]

if itens:
    print("tem coisa na lista")

itens = []

if itens:
    print("tem coisa na lista")  # não aparece

print("sempre aparece")

```

Rode. A primeira vez aparece a primeira linha e a última. A segunda vez só a última.

As regras de verdade e mentira em Python:

```

falso ..... False, 0, "", [], {}, None, ()
verdadeiro ..... qualquer outra coisa

```

`""` (texto vazio) é falso. `0` é falso. `None` é falso. Isso vai aparecer bastante nos próximos capítulos.

Programa completo: nota com conceito e recuperação

Juntando o que já vimos:

```

notas = [9.5, 7.2, 4.0, 6.8]

aprovados = []
reprovados = []

for nota in notas:
    if nota >= 7.0:
        aprovados.append(nota)
    elif nota >= 5.0:
        reprovados.append(nota)

print(f"Aprovados: {len(aprovados)}")
print(f"Em recuperação: {len(reprovados)}")

if not aprovados:
    print("Ninguém passou. Isso é grave.")
elif len(aprovados) == 1:
    print("Passou quase ninguém.")
else:
    print("Passou a maioria.")

```

Três decisões aninhadas no final. Você já sabe ler cada uma.

E o `if not aprovados:` é `truthiness` de novo: `not []` é verdade quando a lista está vazia. Leia em voz alta: "se **não** há aprovados".

Dois erros que todo mundo comete

Erro 1: comparar string com número

```
nota = "7.5"          # texto, porque veio de input sem float()

if nota >= 7.0:       # TypeError!
    print("aprovado")
```

PYTHON

TypeError: '>=' not supported between instances of 'str' and 'float'

Tradução: "não sei comparar texto com número usando `>=`".

A correção é na origem, não no `if`:

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

PYTHON

Como o capítulo 0.2 falou: **depois de** `input`, **sempre** `int()` **ou** `float()`. Esse erro é o mesmo de lá, visto de outro ângulo.

Erro 2: o `=` dentro do `if`

```
if status = "ativo":    # SyntaxError
```

PYTHON

Já falamos. Mas vale reforçar porque acontece em produção: em Django, `if user = request.user:` dá **SyntaxError** na hora, e a mensagem do Python sugere `==`. Nos dois casos o Python te salvou.

Exercícios

Exercício 1 — par ou ímpar

Peça um número e diga se é par ou ímpar. Dica: um número é par quando o resto da divisão por 2 é zero. Em Python, o resto é `%`.

► Resposta

Exercício 2 — três faixas etárias

Peça a idade e classifique: `menor` (até 12), `adolescente` (13 a 17), `adulto` (18 ou mais).

► Resposta

Exercício 3 — o `if` que não está onde você pensa

Este programa **não faz o que o nome sugere**. Descubra por quê antes de corrigir:

```
desconto = 0.10
total = 100.00

if total > 50.00:
    total = total * desconto
    print("Desconto aplicado")

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

PYTHON

► Resposta

No próximo: `for` — como fazer a mesma coisa mil vezes. É a peça que transforma o programa do capítulo 0.3 em algo que realmente processa uma lista inteira.