

PYTHON one

Do zero ao
CRUD com
FastAPI...

Menos teoria
vazia.
Mais código
que funciona.



Thiago Nogueira

Python One — Prévia

Do primeiro print à API REST em produção

Thiago Nogueira

PENSARBEM

Python One — Prévia

© 2026 Thiago Nogueira

Pensarbem

Prévia · 1ª edição

Composto em IBM Plex Serif, IBM Plex Sans e IBM Plex Mono. Diagramado por uma pipeline de tipografia programática.

Sumário

Antes de começar	1
A cooperativa, a empresa e a data	3
Quem aparece	4
Como o código aparece	4
1 O Python que você ouviu falar	9
1.1 Um hobby, no Natal de 1989	12
1.2 Os marcos que ainda afetam o seu código	13
1.3 De onde veio a fama, e o que foi feito de cada coisa	14
1.4 O que mudou de verdade	15
1.5 O que o Python resolve bem	16
1.6 E o que ele não resolve	17
1.7 Quem usa	17
1.8 Conferir a versão é a primeira pergunta	18
2 O que vamos construir	19
2.1 A pergunta que trava o projeto	22
2.2 Três pessoas querendo o mesmo preço	23
2.3 Como um programa pede uma coisa a outro	24
2.4 O que a planilha já é, sem ser	25
2.5 Instalando	25
3 O primeiro programa	27
3.1 O que aconteceu entre o Enter e a saída	29
3.2 O REPL: um Python que responde na hora	29
3.3 Indentação não é estética	29
3.4 Variáveis, sem cerimônia	29
3.5 <code>f-string</code> : o jeito de montar texto	29
3.6 Comentários	29
3.7 Três erros que você vai ver hoje	29
4 Variáveis e tipos	30
4.1 Os cinco tipos do dia a dia	33
4.2 <code>None</code> não é zero, nem vazio, nem falso	33
4.3 O caso do dinheiro	33
4.4 Conversão explícita	33
4.5 Texto é imutável	33
4.6 <code>==</code> compara valor, <code>is</code> compara identidade	33
4.7 Nomes	33
5 Operadores	34

5.1	Aritmética: a divisão é o ponto	37
5.2	Atribuição composta	37
5.3	Comparação, e uma forma que só o Python tem	37
5.4	<code>and</code> , <code>or</code> , <code>not</code> — e o que eles realmente devolvem	37
5.5	O que é verdadeiro	37
5.6	<code>in</code> : a pergunta mais barata que existe	37
5.7	O ternário, escrito ao contrário	37
5.8	Precedência, e a saída honesta	37
5.9	O operador morsa	37
6	Condicionais	38
6.1	O caminho que você não escreveu	41
6.2	A escada de <code>elif</code>	41
6.3	Tratar o caso ruim primeiro	41
6.4	<code>match</code> : quando a pergunta é “qual destes?”	41
6.5	Qual usar	41
6.6	<code>pass</code> , e por que ele existe	41
6.7	A ausência primeiro	41
7	Repetições	42
7.1	<code>range</code> : quando o assunto é o número	45
7.2	<code>enumerate</code> e <code>zip</code>	45
7.3	O acumulador	45
7.4	<code>while</code> : repetir enquanto	45
7.5	Sair no meio: <code>break</code> e <code>continue</code>	45
7.6	O <code>else</code> do laço	45
7.7	Quando o laço deveria ser uma expressão	45
7.8	Sobre laço aninhado	45
8	Funções	46
8.1	Toda função devolve alguma coisa	49
8.2	Argumentos por nome	49
8.3	A armadilha do padrão mutável	49
8.4	<code>*args</code> e <code>**kwargs</code>	49
8.5	Escopo: onde um nome existe	49
8.6	Docstring: a documentação que não desatualiza sozinha	49
8.7	Anotar o tipo, sem ilusão	49
9	Estruturas de dados	50
9.1	Lista: a coleção padrão	53
9.2	Tupla: o registro fixo	53
9.3	Dicionário: a estrutura da API	53
9.4	Conjunto: o que já apareceu	53
9.5	Comprehensions	53
9.6	Duas variáveis, uma lista	53
9.7	Escolhendo	53

10	Módulos e ambiente	57
10.1	Módulo é arquivo	59
10.2	Pacote é pasta	59
10.3	<code>__name__</code> , finalmente	59
10.4	Como o Python encontra um módulo	59
10.5	Importação circular	59
10.6	A biblioteca padrão	59
10.7	O ambiente da cooperativa, não o da máquina	59
10.8	Dependências: instalar é fácil, repetir é o problema	59
10.9	Configuração fora do código	59
11	Classes e objetos	60
11.1	A classe	63
11.2	<code>self</code> não é mágica	63
11.3	Atributo de instância e atributo de classe	63
11.4	Imprimir um objeto	63
11.5	Comparar objetos	63
11.6	Método que muda o estado	63
11.7	Quando a classe não vale a pena	63
12	Encapsulamento	64
12.1	A placa: um sublinhado	67
12.2	Dois sublinhados: uma coisa diferente do que parece	67
12.3	<code>@property</code> : um atributo que é uma função	67
12.4	Propriedade calculada	67
12.5	Invariante: o que nunca pode deixar de ser verdade	67
12.6	A superfície pública	67
13	Herança e protocolos	68
13.1	Herança, em quatro linhas	71
13.2	Sobrescrever não é “melhorar”	71
13.3	O problema da herança: parentesco é para sempre	71
13.4	Duck typing: o contrato sem declaração	71
13.5	Quando o contrato precisa ser escrito	71
13.6	A ordem de resolução	71
14	Exceções	72
14.1	A hierarquia	75
14.2	O <code>except</code> que apaga a evidência	75
14.3	<code>else</code> e <code>finally</code>	75
14.4	<code>with</code> : o <code>finally</code> que você não escreve	75
14.5	Levantar a sua própria exceção	75
14.6	<code>raise ... from</code> : não perder a causa	75
14.7	Pedir perdão ou pedir permissão	75
14.8	Registrar, não imprimir	75
15	Tipagem e dataclasses	76

15.1	O que a anotação faz e o que não faz	79
15.2	A gramática dos tipos	79
15.3	O verificador de tipos	79
15.4	<code>dataclass</code> : o tipo de dado sem cerimônia	79
15.5	<code>Enum</code> : o conjunto fechado de opções	79
15.6	Como o FastAPI usa tudo isso	79
16	O que é o FastAPI	83
16.1	O que o framework decide por você	85
16.2	WSGI e ASGI: a fronteira entre servidor e aplicação	86
16.3	De que o FastAPI é feito	87
16.4	Comparado com o que existia	88
16.5	Oito linhas	88
16.6	Subir, recarregar, desligar	89
16.7	As três páginas que vêm de graça	90
17	O primeiro projeto	92
17.1	A estrutura	95
17.2	As dependências	95
17.3	Configuração: o ambiente, não o código	95
17.4	<code>APIRouter</code> : rotas fora do <code>main</code>	95
17.5	O detalhe do caminho vazio	95
17.6	A ordem das rotas importa	95
17.7	Middleware: o que acontece em toda requisição	95
17.8	CORS, para quando houver um navegador	95
18	A primeira API	96
18.1	O esqueleto	99
18.2	Listar	99
18.3	Buscar um: parâmetro de caminho	99
18.4	Parâmetros de query	99
18.5	Criar: o corpo da requisição	99
18.6	Atualizar e apagar	99
18.7	<code>PUT</code> e <code>PATCH</code> não são a mesma coisa	99
18.8	A lista na memória é mentira	99
19	Modelando com Pydantic	100
19.1	O que aconteceu	103
19.2	A resposta que o cliente recebe	103
19.3	Restrições nos tipos	103
19.4	Tipos que carregam regra	103
19.5	Validador próprio	103
19.6	Entrada e saída são modelos diferentes	103
19.7	Convertendo objetos do banco	103
19.8	Apelidos: quando o JSON não fala a sua língua	103
19.9	Serializar	103

20	Banco de dados	107
20.1	Instalando	109
20.2	A tabela	109
20.3	NOT NULL é uma decisão de negócio	110
20.4	SQL: o mínimo que você precisa entender	110
20.5	Índice: por que a busca fica lenta	111
20.6	Transação: tudo ou nada	112
20.7	Conectando do Python	113
21	SQLAlchemy	116
21.1	Instalar e conectar	119
21.2	A tabela vira classe	119
21.3	A sessão	119
21.4	Escrever	119
21.5	Ler	119
21.6	Alterar e apagar	119
21.7	Ver o SQL gerado	119
22	Repositório	120
22.1	O contrato	123
22.2	A implementação	123
22.3	Três coisas que o repositório não faz	123
22.4	Não devolva <code>Query</code> , devolva dados	123
22.5	A implementação em memória	123
22.6	O custo honesto	123
23	Serviço	124
23.1	A regra sem casa	127
23.2	O serviço	127
23.3	A rota fica curta	127
23.4	A fronteira da transação	127
23.5	Onde o rollback acontece	127
23.6	Regra no serviço ou no banco?	127
24	O CRUD completo	128
24.1	Como as peças se encontram	131
24.2	As rotas	131
24.3	O que faltava no serviço	131
24.4	O esquema de saída	131
24.5	Criando as tabelas e subindo	131
24.6	O passeio completo	131
25	Esquemas	135
25.1	A família de esquemas	137
25.2	Ausente e nulo não são a mesma coisa	137
25.3	Esquemas aninhados	137

25.4	Apelidos e convenções de nome	137
25.5	Exemplos na documentação	137
25.6	O que quebra um contrato	137
26	Validação	138
26.1	Camada 1: o esquema valida a forma	141
26.2	Normalizar é parte de validar	141
26.3	Camada 2: o serviço valida o contexto	141
26.4	Camada 3: o banco garante	141
26.5	Validação entre campos	141
26.6	A mensagem que o cliente recebe	141
26.7	Validar a saída também	141
26.8	Onde a validação não deve estar	141
27	Tratamento de erros	142
27.1	Um tratador por exceção de domínio	145
27.2	O formato do erro	145
27.3	Padronizando o 422	145
27.4	O 500: o único erro que é seu	145
27.5	Identificador de correlação	145
27.6	Que status para quê	145
28	Paginação	146
28.1	O envelope	149
28.2	Os parâmetros	149
28.3	O serviço e o repositório	149
28.4	A ordenação precisa ser total	149
28.5	O custo do OFFSET	149
28.6	Paginação por cursor	149
28.7	Link e X-Total-Count	149
29	Filtros e buscas	150
29.1	O objeto de filtro	153
29.2	Montando a consulta em partes	153
29.3	A ordenação	153
29.4	LIKE, ILIKE e o problema do acento	153
29.5	Quando o LIKE não basta	153
30	Relacionamentos	157
30.1	Um para muitos	159
30.2	O problema N+1	159
30.3	Carregamento preparado	159
30.4	A defesa: proibir o carregamento tardio	159
30.5	Muitos para muitos	159
30.6	cascade: o que acontece com os filhos	159
30.7	O relacionamento no esquema de resposta	159

31	Migrações	160
31.1	Alembic	163
31.2	Gerar e revisar	163
31.3	O que o autogenerate não vê	163
31.4	Renomear sem perder dado	163
31.5	Migração de dados	163
31.6	Alteração sem derrubar a aplicação	163
31.7	Regras de convivência	163
32	Injeção de dependência	164
32.1	Decorador: uma função que recebe uma função	167
32.2	Depends: o framework monta o que a função pediu	167
32.3	Dependência com yield: abrir e fechar	167
32.4	Encadeamento	167
32.5	Dependência que não devolve nada	167
32.6	Dependência como classe	167
32.7	A recompensa: substituir em teste	167
33	Assíncrono	168
33.1	O laço de eventos	171
33.2	O pecado capital: travar o laço	171
33.3	O que o FastAPI faz com def e com async def	171
33.4	Chamar outro serviço	171
33.5	Banco de dados assíncrono	171
33.6	Quando vale a troca	171
34	Autenticação	175
34.1	Senha nunca é guardada	177
34.2	Gerando e conferindo	177
34.3	O modelo de usuário	177
34.4	O esquema: a senha só entra	177
34.5	Cadastro	177
34.6	Login	177
34.7	Sessão ou token	177
35	JWT	178
35.1	A anatomia	181
35.2	Emitir	181
35.3	Verificar	181
35.4	A dependência que identifica	181
35.5	O problema da revogação	181
35.6	Renovação	181
35.7	Onde o cliente guarda	181
35.8	A chave secreta	181
36	Usuários e permissões	182

36.1	Papéis	185
36.2	Exigir papel	185
36.3	401 e 403	185
36.4	A terceira camada: o recurso é dele?	185
36.5	Filtrar a listagem	185
36.6	Registrar quem fez	185
37	Testando Python	189
37.1	pytest	191
37.2	Arrumar, agir, afirmar	191
37.3	Testar o que deve falhar	191
37.4	Fixtures	191
37.5	Parametrização	191
37.6	O que testar	191
37.7	Cobertura	191
37.8	Como rodar	191
38	Testando serviços	192
38.1	O falso	195
38.2	Os tipos de dublê	195
38.3	Quando o simulacro é a resposta certa	195
38.4	Testar o que o serviço decide, não como	195
38.5	Testando o tempo e o acaso	195
38.6	A pirâmide	195
39	Testando a API	196
39.1	TestClient	199
39.2	Substituindo as dependências	199
39.3	O CRUD pelo contrato	199
39.4	Rotas protegidas	199
39.5	Testar o esquema, não só a rota	199
40	Testando o banco	200
40.1	SQLite não serve	203
40.2	O banco de teste	203
40.3	Isolamento por transação	203
40.4	Testando o que só o banco garante	203
40.5	Contêiner: o banco que nasce com a suíte	203
40.6	Esquema por <code>create_all</code> ou por migração?	203
40.7	Testando concorrência	203
41	OpenAPI	207
41.1	O que já vem de graça	209
41.2	Metadados da aplicação	209
41.3	Documentando a rota	209
41.4	Exemplos	209
41.5	Marcando o que vai sair	209

41.6	Versionar	209
41.7	Gerar cliente	209
41.8	A documentação em produção	209
42	Git	210
42.1	O começo	213
42.2	.gitignore, antes de tudo	213
42.3	Commits que contam uma história	214
42.4	Ramos	215
42.5	Conflito	215
42.6	Ganchos de pré-commit	216
42.7	Integração contínua	217
42.8	Comandos que salvam	218
43	Projeto final	221
43.1	Os requisitos	223
43.2	O que o texto não diz	223
43.3	O modelo	223
43.4	Ordem de construção	223
43.5	Empacotando	223
43.6	Subir com as migrações	223
43.7	A rota de saúde, agora a sério	223
43.8	Log em produção	223
43.9	A lista antes de publicar	223
43.10	O ciclo se fecha	223
43.11	Fora do escopo, por enquanto	223
	Respostas	224
	Glossário	232

Antes de começar

Um notebook de 2021, um post-it dizendo para não desligar, e um preço de tomate que dobra se alguém rodar a célula errada.

Esta é a pasta do computador do galpão, o GALPAO-02, que ninguém desliga desde março de 2021:

```
1 fechamento_FINAL_v7.ipynb
2 fechamento_FINAL_v7_BACKUP.ipynb
3 fechamento_NAO_USAR.ipynb
4 precos.py
5 precos_novo.py
6 precos2_NOVO_final.py
7 requirements.txt
8 SABIA_PRECOS_OFICIAL_FINAL.xlsx
```

O `requirements.txt` tem três linhas. Duas estão comentadas. Nenhuma fixa versão. A planilha tem trinta e uma abas e é editada por duas pessoas ao mesmo tempo. O notebook importa `pandas` para abrir essa planilha, calcular a margem e cuspir o preço que a Dona Neuza cola na caixa.

E funciona.

Quatro anos de feira, duzentos produtores, o fechamento da manhã saindo antes das sete. A Dona Neuza abre o galpão às seis, pesa, confere e fecha às cinco. Nesses quatro anos o preço saiu errado um punhado de vezes, e todas as vezes alguém rodou uma célula que tinha um comentário em vermelho pedindo para não rodar.

Você vai substituir essa pasta. Não porque ela é ruim — porque ela só roda naquele computador, na ordem em que o Cacá lembrava de executar as células, e não tem mais para onde crescer.

A cooperativa, a empresa e a data

A **Cooperativa Sabiá** reúne duzentos produtores de hortifrúti. A **Oficina Leme** — vinte e oito pessoas, o tipo de software house que vive de sistema que a planilha não aguenta mais — assinou o contrato para trocar o notebook por um programa que outra empresa consiga chamar.

E existe uma data que ninguém pode empurrar.

A **Rede Bem-Te-Vi**, onze lojas, renova o fornecimento em **15 de abril**. Sem um jeito de consultar preço e estoque sem abrir a planilha, eles compram da cooperativa de baixo. O fornecimento é R\$ 1,2 milhão no ano. O contrato com a Leme é R\$ 74 mil, preço fechado.

Nada disso é cenário. Prazo que não se move, orçamento pequeno em cima de um contrato grande, requisito que chega na sexta e um computador de galpão que precisa continuar fechando o dia enquanto

o substituto é construído — é isso que transforma escolha técnica em decisão. Optar entre duas formas de escrever a mesma coisa só fica interessante quando uma delas custa a manhã da Dona Neuza.

Quem aparece

Bia tem seis anos de carreira, dois deles na Leme, e é a voz que explica o porquê. Está no projeto porque foi a única que aceitou abrir o notebook antes de estimar.

Elias está na Leme há oito anos. Responde olhando para a tela, em geral sem parar de andar, e já aprendeu que resposta precisa vira tarefa com o nome dele.

Rafa é estagiário, quarto período. Faz a pergunta que a sala inteira estava evitando, porque é a única pessoa que não perde nada ao dizer que não entendeu.

Dona Neuza comanda o galpão há trinta e um anos. Sabe de cor a margem de cada produtor e nunca escreveu nenhuma. O comentário `# NÃO RODAR` foi ela quem ditou. O Cacá só digitou.

Helena administra o prazo. O trabalho real dela é receber 15 de abril de cima e reemitir a data para baixo em forma de sprint.

Sérgio é diretor da Leme. Nunca escreveu Python e não finge que escreveu. Volta de palestra com uma caixa nova para o slide.

Cacá é o sobrinho que escreveu o notebook em 2021, aos vinte e três anos, num fim de semana que virou quatro anos. Mudou de estado. Quando alguém pergunta por que a margem está daquele jeito, a resposta da sala é sempre a mesma: o Cacá sabe.

E o **GALPAO-02**, com o post-it, é o sistema em produção. Ele não é o vilão. Cada coisa nova que aparecer aqui vai ser medida contra ele — e em algumas dessas medições o notebook ganha.

Como o código aparece

Código aparece assim, às vezes com o nome do arquivo:

PRECO.PY	python
<pre>preco = 19.90 print(f"R\$ {preco:.2f}")</pre>	

O que o terminal responde aparece sem nome de arquivo e sem realce:

R\$ 19.90

E quando o programa quebra — o que vai acontecer muito, de propósito — o erro vem inteiro, do começo ao fim, porque as linhas do meio são as que importam.

- **O ESSENCIAL**

Comando de terminal aparece com \$ na frente. O \$ representa o prompt e não faz parte do comando: não digite.

Você não precisa instalar nada para começar a ler. Quando o primeiro programa precisar rodar, a instalação vem junto, com o teste que confirma que deu certo.

A data que ninguém pode empurrar é 15 de abril. Até lá, a Dona Neuza abre às seis e o notebook fecha o dia.

PARTE 1

Começando do zero

De onde a linguagem veio, por que a indentação é sintaxe, e o primeiro programa que roda fora da máquina do galpão.

NESTA PARTE

- 1 O Python que você ouviu falar
- 2 O que vamos construir
- 3 O primeiro programa
- 4 Variáveis e tipos
- 5 Operadores
- 6 Condicionais
- 7 Repetições
- 8 Funções
- 9 Estruturas de dados

CAPÍTULO

1

O Python que você ouviu falar

Uma linguagem escrita nas férias de Natal para substituir outra, e que hoje fecha o preço do tomate num computador que ninguém desliga.

“Em dezembro de 1989, eu estava procurando um projeto de programação de ‘hobby’ que me ocupasse na semana do Natal.”

GUIDO VAN ROSSUM, NO PREFÁCIO DE PROGRAMMING
PYTHON, 1996

AO FIM DESTE CAPÍTULO

Saber de onde vem o Python, o que causou a fama de linguagem de script, o que a ruptura entre as versões 2 e 3 ainda cobra, o que a linguagem resolve bem e o que ela não resolve.

ENQUANTO ISSO, NA EMPRESA Não dava pra deixar no Excel?

O Sérgio tinha visto uma palestra na quinta e voltou com três frases na sexta.

— Esse projeto da cooperativa. Por que Python?

— O fechamento é Python — disse a Bia. — A planilha o notebook abre com Python. A Dona Neuza usa isso desde 2021.

— Mas a gente não ia modernizar?

— Vamos. De um notebook que só roda numa máquina para um programa que a Bem-Te-Vi consiga chamar.

O Sérgio fez o gesto de quem tira uma coisa da frente do rosto.

— Não dava pra deixar no Excel e colocar uma IA em cima?

— Dava.

— Então.

— Aí a gente joga fora quatro anos de margem que não estão escritas em lugar nenhum, e entrega em 15 de abril.

Pausa.

— É que Python é meio... — ele procurou a palavra — ...de script.

— A versão atual saiu em outubro.

A Helena falou sem levantar os olhos da planilha de sprint.

— O fechamento da manhã roda em quê?

Elias pigarreou.

— Python.

— Qual versão?

— Três ponto seis. Na máquina que tem um post-it pedindo para não desligar.

Essa conversa acontece em algum lugar do mundo toda semana, e o mal-entendido é sempre o mesmo: “linguagem de script” descreve, com precisão, o motivo pelo qual o Python foi criado — e quase nada do que ele é capaz de carregar hoje.

Os defeitos que alimentam a fama foram reais. Um deles durou uma década e tinha data para acabar. Outro ainda não acabou. Vale saber qual é qual, porque o GALPAO-02 está cheio dos dois, e porque “por que Python?” vai voltar na próxima palestra a que o Sérgio assistir.

1.1 Um hobby, no Natal de 1989

Guido van Rossum era pesquisador no CWI, em Amsterdã. O centro fecharia na semana do Natal. Ele queria um projeto pequeno: um sucessor da linguagem ABC, que ele tinha ajudado a fazer e que era boa para ensinar e ruim para falar com o sistema operacional.

ABC não usava chaves. O bloco era o recuo da linha. Guido levou isso junto, de propósito. A indentação do Python não é uma excentricidade estética: é herança de uma linguagem de ensino que se recusava a ter duas verdades, a do desenho na tela e a do que o computador executava.

O nome também não é o da cobra. Ele estava lendo os roteiros do *Monty Python's Flying Circus* e achou que “Python” era um título curto. Os exemplos clássicos da documentação se chamam `spam` e `eggs` por causa disso. O logotipo da cobra veio depois, quando a comunidade precisou de um desenho.

★ VOCÊ SABIA

Em 20 de fevereiro de 1991 ele publicou a versão 0.9.0 no grupo `alt.sources`.

Não era um anúncio de empresa. Era um arquivo, num fórum, com classes, exceções, funções, listas e dicionários já dentro.

Quem baixa o Python hoje está na continuidade direta daquele arquivo.

A linguagem cresceu do mesmo jeito que o notebook do Cacá: alguém precisava de uma coisa, a coisa entrava, e o projeto deixava de ser hobby sem nunca ter tido uma reunião para decidir isso. A diferença é que, a partir de um certo ponto, as entradas passaram a ter número, discussão pública e data.

1.2 Os marcos que ainda afetam o seu código

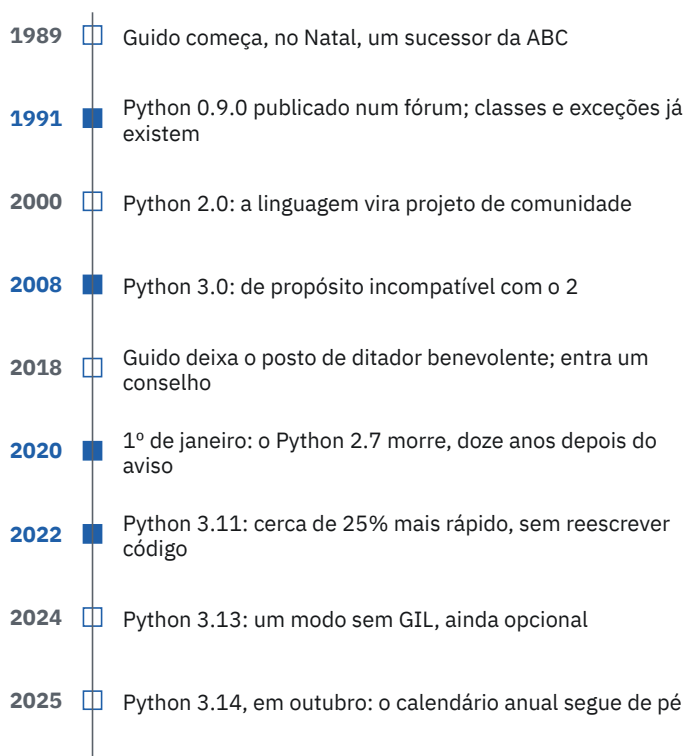


FIGURA 1.1 Por que a linguagem é assim: os marcos que você encontra em código real.

Repare em duas datas.

2008 é quando o Python decidiu quebrar o próprio passado. O texto deixou de ser uma sequência de bytes ambígua e passou a ser texto de verdade. O `print` deixou de ser uma declaração (`print "oi"`) e virou uma função (`print("oi")`). A divisão `7 / 2`, que no Python 2 devolvia `3`, passou a devolver `3.5`. Nenhuma dessas mudanças é detalhe. Código antigo quebra, e o `GALPAO-02` está em 3.6 exatamente porque ninguém quis descobrir em qual célula ele quebraria.

2020 é o dia em que a versão antiga parou de receber correção. Não foi surpresa: o aviso tinha doze anos. Quem ainda roda Python 2 — e ainda há quem rode — não está numa escolha técnica. Está num computador que ninguém teve coragem de desligar.

★ VOCÊ SABIA

Durante anos Guido teve o título oficial de *Benevolent Dictator For Life*, ditador benevolente vitalício. Em 2018 ele saiu do posto depois de uma briga feia sobre um operador novo, o `:=`, que entrou na linguagem com gente demais achando que não devia entrar.

O cargo não foi preenchido. No lugar ficou um conselho eleito. A linguagem que tinha nascido de uma pessoa passou a mudar por votação, com o voto registrado. Isso importa mais do que o operador.

1.3 De onde veio a fama, e o que foi feito de cada coisa

A reputação não foi preconceito. Teve causas concretas. Elas estão aqui para você reconhecê-las quando abrir um notebook de 2021, e para saber, de cada uma, se ela ainda descreve a linguagem que você vai instalar.

A ruptura entre o 2 e o 3. De 2008 a 2020 existiram duas linguagens chamadas Python, incompatíveis, as duas vivas. Tutorial misturava as duas. `python` e `python3` no mesmo Linux eram programas diferentes. Bibliotecas importantes demoraram anos para atravessar. Quem aprendeu naquela década aprendeu a desconfiar da linguagem, e a desconfiança era justa.

O empacotamento. Instalar a biblioteca de outra pessoa foi, por muito tempo, um projeto à parte: `easy_install`, `pip`, `conda`, `venv`, `poetry`, arquivos que não reproduziam a máquina do colega. Isso não foi resolvido com uma versão. Melhorou — e continua sendo o canto mais fácil de se machucar. O `requirements.txt` do galpão, com duas linhas comentadas e nenhuma versão, é esse canto.

O notebook que virou produção. O Jupyter, herdeiro do IPython, foi feito para explorar: rodar uma célula, ver o número, mudar, rodar de novo. A ordem em que as células estão no arquivo não é necessariamente a ordem em que foram executadas. Isso é ótimo para investigar e péssimo para fechar o preço da manhã. Ninguém “lançou” o notebook do Cacá. Ele foi sendo usado até que desligar o computador virou incidente.

O erro que só aparece quando acontece. O Python não recusa o programa antes de rodar por causa de um tipo trocado. `"total: " + 120` é um arquivo válido. Ele quebra na hora em que aquela linha executa

— que, num fechamento, pode ser a primeira terça do mês. A fama de “fácil de começar, difícil de manter” nasce exatamente daí, e ela continua verdadeira para quem não escreve o resto.

A CAUSA	O QUE ACONTECEU COM ELA
duas linguagens chamadas Python	o 2.7 morreu em 1º de janeiro de 2020
<code>print</code> sem parêntese, divisão que truncava	acabaram no Python 3, em 2008
instalar biblioteca como trabalho manual	melhorou; ainda é o ponto frágil
notebook usado como sistema	não é defeito da linguagem; é um uso fora do que o formato aguenta
tipo trocado só quebra na hora	continua verdade; ferramenta à parte é que avisa antes

TABELA 1.1 As duas primeiras linhas não descrevem o Python que você vai instalar. As três últimas descrevem o GALPAO-02, e vão continuar descrevendo qualquer projeto que repita as decisões dele.

1.4 O que mudou de verdade

Quatro coisas, e só a primeira é sobre velocidade.

Dá para anotar o tipo. Desde o Python 3.5 você pode escrever que um preço é um número e que uma função devolve outro número. O interpretador, na hora de rodar, ignora a anotação. Quem não ignora é o editor, o verificador, e qualquer biblioteca que resolva ler a assinatura antes de chamar a função. A linguagem continua dinâmica. Ganhou um jeito de declarar a intenção sem deixar de rodar o que já rodava.

Ficou mais rápido sem pedir reescrita. O Python 3.11, em 2022, executa em média cerca de 25% mais rápido que o 3.10, em código que não mudou. Não é o tipo de velocidade de uma linguagem compilada para uma tarefa numérica pesada. É o tipo que se recebe de graça, numa terça, porque alguém passou anos no motor.

O calendário ficou chato, no bom sentido. Desde 2019 sai uma versão em outubro, todo ano, com dois anos de correção e mais alguns de segurança. A 3.12 ainda recebe correção. A 3.9 não. A 3.6 do galpão parou em dezembro de 2021. “Qual versão?” deixou de ser uma pergunta filosófica.

As decisões ficaram públicas. Proposta numerada, discussão aberta, conselho eleito. Dá para saber por que uma coisa entrou e por

que outra foi recusada. Isso não torna cada decisão boa. Torna cada decisão encontrável — e encontrável é o que separa uma linguagem de um notebook.

• O ESSENCIAL

Escolher uma tecnologia é apostar no que ela faz com o próprio erro.

O Python quebrou a compatibilidade uma vez, avisou por doze anos e cumpriu a data. Acelerou o motor sem cobrar reescrita. Trocou o fundador vitalício por um processo. O empacotamento ele ainda deve — e é honesto tratar isso como dívida aberta, não como detalhe resolvido.

1.5 O que o Python resolve bem

Vale ser específico, porque “serve para tudo” não explica nada. Serve para muita coisa, e mal, quando a frase é essa.

Dá para ler no ano que vem. A indentação obrigatória, a pouca pontuação e o hábito de uma forma óbvia por tarefa existem para que outra pessoa — ou você, em abril — consiga ler o arquivo sem um ritual. O notebook do Cacá é a prova de que a linguagem não obriga ninguém a ser claro. Ela só deixa a clareza mais barata do que em linguagens que precisam de uma página de cerimônia antes da primeira conta.

É boa em falar com os outros. Arquivo, planilha, JSON, banco, página, processo. A biblioteca padrão já sabe data, caminho, texto, JSON e CSV. O trabalho típico não é calcular um milhão de vezes a mesma coisa: é pegar o número de um lugar e entregar em outro, com a regra da cooperativa no meio.

O mesmo idioma cobre o script e o serviço. Não é obrigatório trocar de linguagem no dia em que o `precos.py` passa a atender a rede. Dá para crescer no mesmo arquivo até ele doer, e a dor é que avisa a hora de dividir.

GIL *Global Interpreter Lock*: uma trava que, no Python padrão, deixa apenas uma thread executar código Python de cada vez. Existe para o gerenciador de memória não se perder. Para a maior parte das APIs, o tempo está no banco e na rede, não nessa trava. Ela importa em conta pesada, em paralelo, dentro do processo. Desde o 3.13 existe um modo sem ela, e esse modo não é o que se instala por padrão. Quem sofre com o GIL sabe que sofre. Quem repete a sigla numa reunião, em geral, não mediu.

1.6 E o que ele não resolve

A lista é curta.

Conta numérica pesada, escrita em Python puro. O laço que faz conta o dia inteiro fica lento. A resposta usual não é trocar de linguagem no projeto inteiro: é chamar uma biblioteca que faz aquela conta em C, que é o que `pandas` e o restante da pilha científica já fazem. Usar `pandas` para somar doze células, como o notebook faz, é o extremo oposto — ferramenta de trator para encher um copo.

Interface de celular e de desktop. Não é para isso que a linguagem é escolhida, e forçar é mais caro do que admitir.

Notebook como forma de publicar. Célula com saída gravada, ordem de execução implícita, estado que sobra da run anterior. Ótimo para investigar um número. Ruim para ser o sistema de quem abre o galpão às seis.

A regra que ninguém escreveu. Nenhuma biblioteca, e nenhum modelo de linguagem, sabe que orgânico e agricultura familiar acumulam até quinze por cento exceto na primeira entrega. Isso continua sendo conversa com a Dona Neuza.

1.7 Quem usa

O **Instagram** foi construído em Python. O **Dropbox** também. O **YouTube** começou nele. A **Google** usa a linguagem em escala larga, ao lado de outras. A pilha científica — NumPy, pandas, Jupyter — é o motivo de tanta pesquisa e tanto dado passarem por Python antes de passarem por qualquer outra coisa.

No Brasil ele está onde está o arquivo que alguém precisa processar até amanhã: prefeitura, laboratório, fintech integrando três sistemas que não se falam, e cooperativa cujo fechamento mora num `.ipynb`.

• O ESSENCIAL

“A linguagem mais popular do mundo” é uma frase de ranking. Rankings medem busca, tutorial e resposta de questionário. Não medem sistema em produção nem dinheiro que passa por dentro.

Popularidade não prova qualidade, de nenhuma linguagem. O que ela prova é mercado: existe muito Python escrito, parte dele em notebook, parte dele bom, e alguém vai ter que mantê-lo ou substituí-lo sem perder a regra que só está na cabeça de quem usa.

A qualidade se argumenta pelo que a linguagem tem hoje — leitura barata, calendário público, um jeito de declarar tipos, um ecossistema enorme para falar com o resto do mundo — e pelo contorno honesto do que ela não faz.

1.8 Conferir a versão é a primeira pergunta

O Python 3.6 do galpão não recebe correção desde dezembro de 2021. Isso não é opinião sobre o Cacá. É o calendário.

▶ NA PRÁTICA

Quando o Python estiver instalado, rode `python --version` e guarde o número. Em seguida abra o interpretador e digite:

```
>>> import this
```

Ele imprime o *Zen do Python*, dezenove linhas em inglês, escritas por Tim Peters. Quatro delas bastam por hoje:

*There should be one— and preferably only one —obvious way to do it.
Errors should never pass silently. Unless explicitly silenced.*

A primeira é o critério para escolher entre duas formas de escrever a mesma coisa. O par de baixo é a célula 14: ela não dá erro. Alguém silenciou, com um comentário, e o tomate dobra.

RESUMO

- O Python nasceu em 1989 como sucessor da ABC. A indentação é herança dessa origem, não enfeite.
- De 2008 a 2020 existiram duas linguagens incompatíveis com o mesmo nome. O 2.7 morreu em 1º de janeiro de 2020.
- O que continua verdadeiro: empacotar dependência ainda é frágil, notebook não é forma de publicar, e tipo trocado só quebra quando a linha roda.
- O que mudou: tipos anotados, calendário anual em outubro, processo de decisão público, e um motor mais rápido sem pedir reescrita.
- O 3.6 do galpão está sem correção desde dezembro de 2021.

CAPÍTULO

2

O que vamos construir

Onze caixas num slide, setenta e quatro mil reais, e ninguém na sala capaz de dizer o que a célula 14 faz.

“Existem duas formas de construir um projeto de software. Uma é fazê-lo tão simples que, obviamente, não há deficiências. A outra é fazê-lo tão complicado que não há deficiências óbvias.”

C. A. R. HOARE

AO FIM DESTES CAPÍTULO

Entender o que a Bem-Te-Vi está comprando quando pede “um sistema”, separar planilha de regra de preço, e terminar com o Python instalado e conferido.

ENQUANTO ISSO, NA EMPRESA Sabiá Data Platform

O slide tinha onze caixas ligadas por setas e, no meio, uma caixa maior, escrita **IA**.

— É simples — disse o Sérgio. — A gente plataforma o dado, expõe a API e plugga inteligência no preço.

Ele tinha vendido o projeto ao conselho da cooperativa em quarenta minutos usando essas três palavras, o que é um talento de verdade.

A Bia levantou a mão.

— O que o notebook calcula hoje?

Houve um silêncio de três segundos, que numa reunião de nove pessoas dá tempo de alguém mexer na caneta de propósito.

— Ele... processa — disse o Elias.

— Processa o quê?

O Elias tinha oito anos de Leme e já havia aprendido que resposta precisa vira tarefa com o seu nome. Olhou para a Helena.

— O Cacá sabe.

— O Cacá mudou de estado.

— Ele responde WhatsApp.

— E a gente entrega quando?

A Helena respondeu sem consultar nada, porque administrava datas do jeito que um bombeiro administra fósforo.

— A Bem-Te-Vi renova em 15 de abril.

— E se passar de 15 de abril?

— Eles comprem da cooperativa de baixo. Um milhão e duzentos no ano.

A reunião durou mais cinquenta minutos e produziu três decisões de arquitetura sobre um notebook que ninguém presente conseguia executar.

Naquela mesma tarde, a Bia foi ao galpão ver o que a Leme tinha acabado de se comprometer a substituir.

ENQUANTO ISSO, NA EMPRESA A célula 14

— Você é a do computador? — perguntou a Dona Neuza, sem tirar o olho da balança.

— Sou.

— Então não roda a 14.

O notebook estava aberto na célula com um comentário em vermelho, de 2021: # NÃO RODAR – Cacá.

— O que ela faz?

— Dobra o tomate. Descobri em março, porque o Seu Onofre achou graça do preço. Desde então eu pulo.

— E o resto?

Ela apontou a planilha com o queixo. Trinta e uma abas. Duas com o mesmo nome, uma com _OK no final.

— Quando a menina do escritório e eu abrimos ao mesmo tempo, uma das duas perde a manhã, e a gente só descobre na hora de carregar o caminhão. Quando o produto vem de fora do estado, a margem é outra, e isso está na cabeça, não na célula. Quando alguém busca açafraão sem o acento, não acha. E o de sempre: o número do notebook e o número da planilha não fecham.

— Quanto não fecha?

— Três reais e quarenta. Em doze mil. Todo mês.

— Há quanto tempo?

— O do tomate, desde março. Os três e quarenta eu já nem lembro.

— E ninguém arrumou?

A Dona Neuza colou a etiqueta na caixa, alisou com o polegar e puxou a seguinte.

— Todo ano vem um moço e diz que vai arrumar. Você é a terceira. O Cacá foi o primeiro, e ele ainda deve um sábado.

2.1 A pergunta que trava o projeto

A reunião da manhã não foi um caso de incompetência. Aconteceu o que acontece em empresa de qualquer tamanho: o sistema é um arquivo antigo, quem escreveu foi embora, e a cobrança por um cronograma chega antes do entendimento.

O resultado é sempre o mesmo — decisões tomadas em cima de uma caixa escrita **IA**.

Repare em quem, das nove pessoas da reunião, conseguiu descrever o comportamento do sistema: ninguém. E repare em quanto tempo a Dona Neuza levou para descrever o dela: menos de um minuto, sem parar de etiquetar, com quatro defeitos e uma frequência para cada um.

■ **NA SUA CARREIRA**

“O que esse sistema faz hoje?” é a pergunta mais barata e mais impopular de qualquer projeto que começa com a palavra plataforma. Quando for você a perguntar, peça um **exemplo**, não uma definição: “*me mostra uma coisa que acontece de manhã, do começo ao fim*”. Definição todo mundo improvisa. Exemplo, não. E anote a resposta na frente de quem respondeu. Uma descrição que não foi escrita na hora vira, duas semanas depois, duas descrições diferentes — em geral na véspera de uma sprint.

2.2 Três pessoas querendo o mesmo preço

Antes de decidir qualquer ferramenta, vale entender o que a cooperativa precisa no lugar da pasta.

Hoje o preço mora em dois lugares que não conversam: a planilha e o notebook. Quem não está de pé na frente da Dona Neuza não consulta nada. O comprador da Bem-Te-Vi liga. O Seu Onofre manda um áudio perguntando quanto ficou a caixa de alface. Alguém abre a aba errada.

O que o contrato de R\$ 74 mil precisa entregar, no fundo, são três usos do mesmo número:

QUEM	O QUE PRECISA
a Dona Neuza, na balança	registrar o que chegou, ver o preço, baixar a caixa
o comprador da Bem-Te-Vi	consultar estoque e preço sem ligar para o galpão
o Seu Onofre, no telefone	ver só o que é dele, e quanto a cooperativa ficou devendo

TABELA 2.1 Três perguntas diferentes, um preço só.

São três programas distintos, possivelmente escritos por pessoas distintas. Se cada um calcular a margem do seu jeito, “orgânico acumula com agricultura familiar?” vai existir em três versões — e elas vão divergir em três velocidades diferentes. A de março, a do tomate dobrado, já é uma amostra.

A saída é ter **um** programa que sabe as regras, e combinar um jeito de os outros pedirem as coisas a ele.

Esse jeito combinado tem nome.

API *Application Programming Interface*, interface de programação. É um contrato entre dois programas: um sabe fazer alguma coisa, o outro precisa que ela seja feita, e existe uma forma acordada de pedir e de responder.

A Dona Neuza é uma API há trinta e um anos e ninguém nunca a chamou assim. O Seu Onofre chega no galpão, diz o produto e mostra a nota. Ela responde com o preço, ou com “acabou, volta quinta”. Ele não precisa saber em qual aba está o número, nem qual célula não pode rodar. Precisa saber **o que pedir** e **em que formato ela responde**.

É isso que vamos escrever. A diferença é que o pedido vai chegar pela rede, e a resposta vai sair em texto que outro programa consegue ler.

2.3 Como um programa pede uma coisa a outro

Quando o sistema da Bem-Te-Vi quiser o preço do tomate, ele manda pela rede um bloco de texto parecido com este:

```
GET /produtos/12
Accept: application/json
```

A primeira linha diz **o que fazer** (GET, que significa “me dê”) e **onde** (/produtos/12, o produto número 12). A segunda diz em que formato a resposta é bem-vinda.

Para cadastrar um produto novo, o bloco muda de verbo e ganha um corpo, depois de uma linha em branco:

```
POST /produtos
Content-Type: application/json

{"nome": "Tomate italiano", "preco": "8.90"}
```

O POST é um dos verbos que cobrem quase tudo que a cooperativa faz hoje na planilha, sem chamar nada disso de verbo:

VERBO	SIGNIFICA	NO GALPÃO
GET	me dê	“quanto está o tomate?”
POST	crie um novo	entra produto que a cooperativa ainda não tinha
PUT	substitua	o preço mudou por inteiro
PATCH	altere um pedaço	só o estoque, o resto fica
DELETE	tire	saiu do catálogo

TABELA 2.2 Cinco verbos cobrem o que a planilha faz com trinta e uma abas.

Repare no endereço: /produtos, e não /criarProduto. O verbo já está do lado de fora, na primeira palavra. Quem coloca o verbo dentro do

endereço termina com `/criarProduto`, `/atualizarProduto` e `/apagarProduto` — três endereços para uma coisa só, e nenhum jeito de combinar “tomate” com “abaixo de dez reais”.

A resposta volta com um número de três dígitos na frente. `200` é “ai está”. `201` é “criei”. `404` é “não tenho esse”. `422` é “entendi o pedido e recusei, olha o que está errado”. `500` é “quebrou aqui dentro”. Errar esse número é mentir para o programa que confiou em você: ele toma uma decisão em cima do que você disse que aconteceu.

2.4 O que a planilha já é, sem ser

As quatro operações que a Dona Neuza faz todo dia têm iniciais que viraram palavra: criar, ler, atualizar, apagar. **CRUD**. Não é uma arquitetura. É o nome do trabalho mais comum de quem guarda dado de outra pessoa.

Fazer as quatro coisas é pouco. O notebook já faz as quatro, e mesmo assim dobra o tomate e perde três reais e quarenta. O trabalho é fazê-las sem duas pessoas gravarem em cima uma da outra, sem margem calculada diferente em cada aba, e sem uma célula que só a Dona Neuza sabe pular.

2.5 Instalando

Baixe o Python em <https://python.org>. No Windows, marque **Add python.exe to PATH** na primeira tela. No macOS e no Linux quase sempre já existe um Python instalado, e ele pode ser o 3.6 de alguém: confira antes de confiar.

Abra o terminal e digite:

```
$ python --version
Python 3.12.4
```

Se aparecer `3.12` ou mais alto, está pronto. Se o comando não for encontrado, tente `python3`. Em macOS e Linux esse é o nome usual.

▲ A LOJA DO WINDOWS

Se você instalar o Python pela Microsoft Store, algumas versões deixam um `python.exe` falso no caminho, que abre a loja em vez de rodar o programa. Baixe do site oficial e marque a caixa do PATH.

⚡ ERRO COMUM

`python` e `python3` podem ser programas diferentes na mesma máquina, com versões diferentes. O galpão vive disso: um terminal responde 3.6, outro responde 3.12, e os dois se chamam Python. Descubra qual é o seu agora e use sempre o mesmo. Metade dos “mas eu instalei essa biblioteca” da vida real é esta confusão, e a outra metade é instalar no Python do sistema inteiro — assunto para o dia em que houver uma biblioteca para instalar.

▶ NA PRÁTICA

Rode `python --version` e, em seguida, `python` sem nome de arquivo. O próprio interpretador abre e espera com `>>>`. Digite `2 + 2` e depois `exit()`. Se a resposta foi 4, a ferramenta que o resto do trabalho usa está na mesa. Não instale biblioteca nenhuma ainda. Não há o que instalar: ainda não existe um programa.

RESUMO

- O sistema de hoje é um notebook mais uma planilha, e os dois discordam.
- Uma API é um contrato: o que se pede, onde, e em que formato volta a resposta.
- O verbo fica fora do endereço. O endereço nomeia a coisa.
- O número de três dígitos é a primeira frase da resposta.
- O Python da máquina se confere com `python --version` antes de confiar nele.

▣ O PROJETO AGORA

Você tem o Python instalado e conferido, e sabe o que a Bem-Te-Vi vai perguntar em 15 de abril. Ainda não há programa. É daqui que este começa.

- 1 O Seu Onofre manda um áudio: “Neuza, sobrou alface e quanto está a caixa?” Escreva, em duas linhas, o que ele pediu e o que ela responderia — e marque o que ele **não** precisou saber para receber a resposta.
- 2 Sem olhar a tabela, escreva o verbo e o endereço de cada operação:
 .. cadastrar um produto, listar os produtos, ver um, trocar o preço, tirar do catálogo.
- 3 O Sérgio pediu um endereço `/produtos/baratos`, que devolva o que
 ... custa menos de dez reais. Você aceitaria? Justifique em três linhas.

CAPÍTULO

3

O primeiro programa

*Dois Pythons na mesma manhã, e só um deles
pode fechar o galpão.*

AO FIM DESTE CAPÍTULO

Escrever, salvar e rodar um programa Python; entender o que acontece entre o `Enter` e a saída; e ler as três mensagens de erro que todo iniciante vê.

**Capítulo disponível na
versão completa**

CAPÍTULO

4

Variáveis e tipos

O Python não pergunta o tipo. Isso não quer dizer que ele não se importe.

AO FIM DESTE CAPÍTULO

Usar os tipos básicos com consciência, converter entre eles sem perder dado, saber por que `float` não serve para dinheiro e nomear variáveis como quem escreve para outra pessoa.

**Capítulo disponível na
versão completa**

CAPÍTULO

5

Operadores

Os símbolos são os de sempre. As regras por trás deles é que não são.

AO FIM DESTES CAPÍTULO

Calcular, comparar e combinar condições sem surpresa — inclusive nos três pontos em que o Python se comporta de um jeito que nenhuma outra linguagem popular se comporta.

**Capítulo disponível na
versão completa**

CAPÍTULO

6

Condicionais

Escolher um caminho é assumir o outro — inclusive quando ele não está escrito.

AO FIM DESTA CAPÍTULO

Escrever decisões com `if`, `elif` e `match`, escolher entre eles por intenção, e reconhecer o caminho implícito que todo `if` sem `else` deixa para trás.

**Capítulo disponível na
versão completa**

CAPÍTULO

7

Repetições

Em Python não se conta índice. Se percorre — e essa diferença muda o que você consegue escrever.

AO FIM DESTES CAPÍTULO

Percorrer coleções com `for`, repetir com `while`, sair no meio com `break`, e reconhecer quando um laço deveria ser uma expressão.

**Capítulo disponível na
versão completa**

CAPÍTULO

8

Funções

Dar nome a um pedaço de trabalho é a primeira decisão de arquitetura que alguém toma.

AO FIM DESTA CAPÍTULO

Escrever funções com parâmetros, valores padrão e retorno; entender o escopo dos nomes; e evitar a armadilha de valor padrão mutável, que é a mais famosa da linguagem.

**Capítulo disponível na
versão completa**

CAPÍTULO

9

Estruturas de dados

Quatro coleções e uma pergunta: o que você vai precisar perguntar a esses dados depois?

AO FIM DESTES CAPÍTULOS

Escolher entre lista, tupla, dicionário e conjunto por critério; fatiar, ordenar e transformar coleções; e saber o que acontece quando duas variáveis apontam para a mesma lista.

**Capítulo disponível na
versão completa**

PARTE 2

O Python que você realmente precisa

Não a linguagem toda — a parte dela que sustenta uma API. Módulos, objetos, exceções e tipagem moderna, na dose exata.

NESTA PARTE

- 10 Módulos e ambiente
- 11 Classes e objetos
- 12 Encapsulamento
- 13 Herança e protocolos
- 14 Exceções
- 15 Tipagem e dataclasses

CAPÍTULO

10

Módulos e ambiente

Um arquivo só é um script. Dois arquivos já são uma decisão de arquitetura.

AO FIM DESTES CAPÍTULOS

Dividir código em módulos e pacotes, importar sem ciclos, instalar dependências de forma reprodutível e separar o que é da cooperativa do que é da máquina de quem está programando.

**Capítulo disponível na
versão completa**

CAPÍTULO

11

Classes e objetos

Um dicionário aceita qualquer chave. Um tipo com nome recusa a chave errada — e isso é o serviço que ele presta.

AO FIM DESTES CAPÍTULOS

Criar classes com estado e comportamento, entender `self`, implementar `__repr__` e `__eq__`, e saber quando um dicionário já basta.

**Capítulo disponível na
versão completa**

CAPÍTULO

12

Encapsulamento

Python não tranca a porta. Ele coloca uma placa — e a placa funciona melhor do que quase ninguém espera.

AO FIM DESTES CAPÍTULOS

Proteger a coerência de um objeto com convenção e `@property`, entender por que Python não tem `private`, e projetar a superfície pública de uma classe.

**Capítulo disponível na
versão completa**

CAPÍTULO

13

Herança e protocolos

Em Python, parecer um pato basta. A pergunta é quando isso é liberdade e quando é armadilha.

AO FIM DESTES CAPÍTULOS

Usar herança sem criar parentesco indesejado, preferir composição quando ela cabe, e descrever contratos com `Protocol` e classes abstratas.

**Capítulo disponível na
versão completa**

CAPÍTULO

14

Exceções

Um erro tratado errado é pior que um erro não tratado: ele apaga a evidência.

AO FIM DESTES CAPÍTULOS

Tratar falhas com `try/except` sem engolir informação, criar exceções próprias, usar `with` para liberar recursos e escolher entre pedir permissão e pedir perdão.

**Capítulo disponível na
versão completa**

CAPÍTULO

15

Tipagem e dataclasses

O Python ignora as anotações de tipo em execução. Quase tudo o mais no seu projeto, não.

AO FIM DESTES CAPÍTULOS

Anotar tipos com precisão, criar tipos de dado com `dataclass` e `Enum`, e entender por que o FastAPI consegue validar uma requisição lendo apenas a assinatura de uma função.

**Capítulo disponível na
versão completa**

PARTE 3

Entrando no FastAPI

O framework que lê tipos Python e devolve documentação, validação e rotas prontas. E o preço disso: entender o que ele decide por você.

NESTA PARTE

- 16 O que é o FastAPI
- 17 O primeiro projeto
- 18 A primeira API
- 19 Modelando com Pydantic

CAPÍTULO

16

O que é o FastAPI

Um framework é um monte de decisões já tomadas. Vale saber quais são antes de aceitá-las.

AO FIM DESTE CAPÍTULO

Entender o que um framework web resolve, o que é ASGI, de que peças o FastAPI é feito, e ler linha a linha o menor programa que responde na rede.

ENQUANTO ISSO, NA EMPRESA Já tá no ar?

— A Bem-Te-Vi perguntou se já tem endereço — disse a Helena, na terça. — Mandeí o do galpão. Eles disseram que não abrem notebook.

O Sérgio já tinha atualizado o slide. A caixa **IA** ganhara uma seta para uma caixa nova, **API**, com a palavra *live* embaixo.

A Bia mostrou o que tinha: um programa que abria uma porta, lia bytes e devolvia uma chave. Funcionava para uma frase. Não sabia o que era um produto, não dizia “não achei”, e caía fora se o pedido viesse maior do que ela tinha previsto.

— Isso é a API? — perguntou o Rafa.

— Isso é a parte chata — disse o Elias. — A parte da cooperativa ainda nem começou. E a parte chata, se a gente escrever à mão, a gente vai escrever errado na sexta.

Um servidor HTTP recebe bytes de uma rede e precisa devolver bytes. Entre uma coisa e outra existe uma quantidade enorme de trabalho repetitivo: interpretar o texto da requisição, separar caminho de query, decodificar o corpo, escolher qual função chamar, converter o retorno em JSON, montar os cabeçalhos, escolher o status.

Escrever isso à mão uma vez é instrutivo. Escrever em todo projeto é desperdício. É para isso que existe um framework.

16.1 O que o framework decide por você

SEM_FRAMEWORK.PY

python

```
1  import socket
2
3  servidor = socket.socket()
4  servidor.bind(("127.0.0.1", 8000))
5  servidor.listen()
6
7  while True:
8      cliente, _ = servidor.accept()
9      pedido = cliente.recv(1024).decode()
10     print(pedido.splitlines()[0])
11     cliente.send(b"HTTP/1.1 200 OK\r\n\r\n{ }")
12     cliente.close()
```

Isso é um servidor HTTP. Ele funciona — e só. Não sabe rotear, não lê JSON, atende uma conexão por vez, quebra com requisição maior que 1024 bytes e não tem ideia do que seja um 404.

O framework preenche essa distância. Em troca, ele impõe uma forma de escrever: onde as rotas ficam, como os dados entram, o que a sua função pode devolver.

• O ESSENCIAL

Framework não é biblioteca. Biblioteca é código que **você chama** framework é código que **chama o seu**. Essa inversão é o que dá poder e é também o que dá o incômodo: quando algo não encaixa, você não consegue simplesmente pular a parte que atrapalha.

16.2 WSGI e ASGI: a fronteira entre servidor e aplicação

Em 2003, a comunidade Python padronizou a conversa entre servidor e aplicação num documento chamado WSGI. Graças a ele, qualquer aplicação Django ou Flask roda em qualquer servidor compatível — Gunicorn, uWSGI, mod_wsgi.

O WSGI tem uma limitação de nascença: ele é **síncrono**. Cada requisição ocupa um trabalhador do começo ao fim. Se a requisição espera meio segundo por um banco de dados, o trabalhador fica meio segundo parado, sem poder atender mais ninguém.

O ASGI é o sucessor, e a única mudança conceitual é essa: uma requisição pode **ceder a vez** enquanto espera. O capítulo 33 trata do assunto a sério; por ora, basta o mapa:

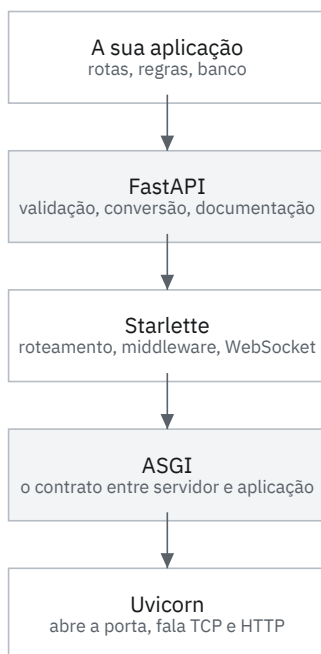


FIGURA 16.1 Cada camada acrescenta decisões já tomadas — e cobra por elas em abstração.

16.3 De que o FastAPI é feito

O FastAPI é, literalmente, duas bibliotecas costuradas:

Starlette cuida do lado HTTP: roteamento, middleware, WebSocket, tarefas de fundo, arquivos estáticos. É um framework ASGI completo e minimalista, e quando você lê na documentação que “o FastAPI herda de Starlette”, é literal.

Pydantic cuida do lado dos dados: lê anotações de tipo, valida, converte, e gera o esquema JSON. É o assunto do capítulo 19.

O FastAPI é a cola entre os dois, e a cola é a parte interessante: ele inspeciona a assinatura de cada função de rota e deduz, de cada parâmetro, se ele vem do caminho, da query, do corpo, do cabeçalho ou de uma dependência.

★ VOCÊ SABIA

O FastAPI nasceu em 2018, escrito por Sebastián Ramírez. A ideia não era nova — vários projetos já tinham tentado usar anotações de tipo para gerar validação. O que estava faltando era o Python 3.6, com anotação de variável, e uma biblioteca de validação madura o bastante. O framework é, em boa medida, o reconhecimento de uma oportunidade que já estava na mesa havia dois anos.

16.4 Comparado com o que existia

FRA-MEWORK	FILOSOFIA	QUANDO ESCOLHER
Django	traz tudo: ORM, admin, templates	sistema com telas e painel
Flask	traz o mínimo; você monta	projeto pequeno, controle total
FastAPI	API tipada, documentação gerada	API para outro programa consumir

TABELA 16.1 Não há vencedor. Há adequação — e uma API REST sem telas é o caso mais claro a favor do FastAPI.

A diferença prática mais visível é a documentação. Em Flask, descrever os endpoints é trabalho manual, feito à parte, que desatualiza. Em FastAPI, a descrição é o código.

16.5 Oito linhas

MAIN.PY

python

```

1  from fastapi import FastAPI
2
3  app = FastAPI()
4
5
6  @app.get("/saude")
7  def saude():
8      return {"status": "de pé"}
```

O QUE CADA PARTE FAZ

```
app = FastAPI()

@app.get("/saude")
def saude():
    return {"status": "de pé"}
```

1

2 3

4

5

- 1 app é a aplicação ASGI: é este objeto que o Uvicorn procura.
- 2 @app.get é um decorador: registra a função na tabela de rotas.
- 3 "/saude" é o caminho; o verbo está no nome do método.
- 4 O nome da função não aparece na URL — ele é só para você.
- 5 Devolver um dicionário basta: o FastAPI converte em JSON e põe 200.

O decorador merece um parágrafo, porque é a construção Python que mais assusta quem vê pela primeira vez. @app.get("/saude") acima de def saude() significa, em Python puro:

```
saude = app.get("/saude")(saude)
```

Ou seja: app.get("/saude") devolve uma função, e essa função recebe a sua saude e faz algo com ela — no caso, guarda numa tabela de rotas. Não há mágica. Há uma função recebendo outra função, e a tabela é o que o servidor consulta quando o pedido chega.

16.6 Subir, recarregar, desligar

```
$ fastapi dev main.py
```

```
Server started at http://127.0.0.1:8000
Documentation at http://127.0.0.1:8000/docs
```

O modo dev liga o recarregamento automático: salvou o arquivo, o servidor reinicia. Isso é ótimo em desenvolvimento e **inaceitável** em produção — o processo que vigia arquivos custa recursos e recarrega o servidor no pior momento possível.

```
$ fastapi run main.py          # produção: sem recarga
$ uvicorn main:app --workers 4
```

Na segunda forma, main:app significa “no módulo main, o objeto app”. É o endereço que o processo de produção precisa: quem sobe o servidor não é mais o modo de desenvolvimento, e alguém tem que dizer qual

objeto atender a porta. `--workers 4` abre quatro processos. Cada um atende sozinho; juntos, atendem quatro pedidos ao mesmo tempo.

⚠ ERRO COMUM

A porta 8000 pode estar ocupada — muitas vezes por um servidor que você esqueceu aberto em outro terminal. O sintoma é

[Errno 98] Address already in use. Suba em outra porta com `--port 8001` ou encerre o processo antigo; o que **não** funciona é rodar duas vezes e supor que a segunda vale.

16.7 As três páginas que vêm de graça

ENDEREÇO	O QUE É
<code>/docs</code>	Swagger UI: documentação navegável e executável
<code>/redoc</code>	a mesma informação, em formato de leitura
<code>/openapi.json</code>	o esquema cru, que gera as duas anteriores

TABELA 16.2 O `/openapi.json` é o que importa: é um padrão, e ferramentas externas geram clientes em outras linguagens a partir dele.

▶ NA PRÁTICA

Abra `/docs`, expanda a rota `/saude` e clique em **Try it out** e depois em **Execute**. A página executa a requisição de verdade e mostra o comando `curl` equivalente. Esse botão substitui, nas primeiras semanas de um projeto, qualquer ferramenta externa de teste de API.

RESUMO

- Framework é código que chama o seu; a inversão é o poder e o incômodo.
- ASGI sucede o WSGI permitindo que uma requisição ceda a vez enquanto espera.
- FastAPI = Starlette (HTTP) + Pydantic (dados) + a leitura das assinaturas.
- `@app.get(...)` é um decorador: uma função que recebe a sua função.
- `fastapi dev` recarrega; produção usa `run` ou Uvicorn com trabalhadores.
- `/docs`, `/redoc` e `/openapi.json` vêm do próprio código.

- 1 Acrescente uma rota `GET /versao` que devolva `{"versao": "0.1.0"}` e confira no `/docs` que ela apareceu sozinha na documentação.
- 2 Crie duas rotas com o mesmo caminho e verbos diferentes — `GET /ping` e `POST /ping` — e explique por que isso não é conflito.

- 3** Sua equipe precisa escolher entre Django e FastAPI para um sistema que tem uma API pública consumida por um aplicativo, um painel administrativo interno, e relatórios em PDF. Escreva a recomendação em cinco linhas.
- ...

CAPÍTULO

17

O primeiro projeto

Um arquivo `main.py` de mil linhas nunca foi uma decisão. Foi a ausência de uma.

AO FIM DESTES CAPÍTULO

Montar a estrutura de pastas do projeto, separar rotas com `APIRouter`, carregar configuração de variáveis de ambiente e deixar tudo rodando com um comando.

**Capítulo disponível na
versão completa**

CAPÍTULO

18

A primeira API

Cinco rotas, uma lista na memória, e a primeira vez que outro programa consegue usar o seu.

AO FIM DESTES CAPÍTULOS

Escrever o CRUD completo com armazenamento em memória, receber parâmetros de caminho, de query e de corpo, e devolver os códigos de status certos.

**Capítulo disponível na
versão completa**

CAPÍTULO

19

Modelando com Pydantic

A mesma classe que descreve o dado também o recusa quando ele chega errado.

AO FIM DESTE CAPÍTULO

Substituir `dict` por modelos Pydantic, validar entrada com tipos e restrições, escrever validadores próprios e entender o que muda na documentação e nas mensagens de erro.

**Capítulo disponível na
versão completa**

PARTE 4

Construindo o CRUD

A espinha dorsal do livro. Schema, banco, repositório, serviço e rota — nesta ordem, uma camada por vez.

NESTA PARTE

- 20 Banco de dados
- 21 SQLAlchemy
- 22 Repositório
- 23 Serviço
- 24 O CRUD completo

CAPÍTULO

20

Banco de dados

*Um lugar que não esquece, não se contradiz e
aguenta dez pessoas escrevendo ao mesmo
tempo.*

AO FIM DESTES CAPÍTULO

Instalar o PostgreSQL, modelar uma tabela, escrever SQL suficiente para entender o que o ORM vai gerar, e saber o que uma transação garante.

A lista do capítulo 18 some quando o servidor reinicia. A primeira reação de todo mundo é gravar num arquivo — e vale entender por que essa reação está errada antes de instalar coisa nenhuma.

Um arquivo JSON resolve a persistência e nada mais. Duas requisições simultâneas sobrescrevem uma à outra. Buscar um produto exige ler o arquivo inteiro. Não existe “gravar as duas coisas ou nenhuma”. E qualquer falha no meio da escrita deixa um arquivo pela metade, que na próxima leitura não carrega.

O banco de dados relacional resolve esses quatro problemas há cinquenta anos.

20.1 Instalando

A forma mais rápida, se você tem Docker:

```
$ docker run --name pg-sabia -e POSTGRES_PASSWORD=senha \
  -e POSTGRES_DB=catalogo -p 5432:5432 -d postgres:16
```

Sem Docker, baixe o instalador em <<https://postgresql.org/download>>. No macOS, o Postgres.app é a opção com menos atrito.

```
$ psql "postgresql://postgres:senha@localhost/catalogo"
psql (16.4)
catalogo=#
```

▶ NA PRÁTICA

Rode `\dt` para listar tabelas (nenhuma ainda), `\l` para listar bancos e `\q` para sair. Esses três comandos resolvem noventa por cento das vezes em que você precisa olhar o banco durante o desenvolvimento — e são mais rápidos que abrir uma ferramenta gráfica.

20.2 A tabela

CRIAR_TABELA.SQL

sql

```
1 CREATE TABLE produto (
2     id          BIGSERIAL PRIMARY KEY,
3     nome        VARCHAR(120) NOT NULL,
4     preco       NUMERIC(10,2) NOT NULL CHECK (preco > 0),
5     estoque     INTEGER      NOT NULL DEFAULT 0 CHECK (estoque
6     >= 0),
7     ativo       BOOLEAN      NOT NULL DEFAULT TRUE,
8     criado_em   TIMESTAMPTZ  NOT NULL DEFAULT NOW()
9 );
```

Cada palavra dessa declaração é uma decisão, e todas voltam no capítulo 21 em forma de código Python.

ESCOLHA	POR QUÊ
BIGSERIAL PRIMARY KEY	identificador único gerado pelo banco
NUMERIC(10,2)	dinheiro exato — nunca FLOAT
NOT NULL	o campo não aceita ausência
DEFAULT 0	valor quando ninguém informa
TIMESTAMP TZ	instante com fuso

TABELA 20.1 NUMERIC guarda decimal em base 10, e é a contraparte no banco do Decimal do capítulo 4.

▲ **ATENÇÃO**

TIMESTAMP sem o TZ guarda uma data e hora sem dizer de onde. Dois servidores em fusos diferentes gravam instantes diferentes com o mesmo texto, e a ordenação por data passa a mentir. Use TIMESTAMPTZ sempre, e guarde em UTC.

20.3 NOT NULL é uma decisão de negócio

A pergunta “este campo pode faltar?” parece técnica e é de negócio. Um produto sem preço existe? Um produtor sem e-mail existe?

Deixar tudo aceitando nulo parece flexível e é o contrário: empurra a pergunta para cada consulta que lê o campo, para sempre. Cada coluna que aceita nulo é um if a mais em todo código que a toca.

● **O ESSENCIAL**

A regra prática: comece com NOT NULL em tudo e remova quando alguém apresentar um caso real de ausência. O caminho contrário — começar permissivo e apertar depois — exige limpar os dados já gravados, e isso nunca é uma migração tranquila.

20.4 SQL: o mínimo que você precisa entender

A consulta abaixo é o que o programa pede ao banco. Vale saber escrevê-la e vale saber lê-la: no dia em que ela estiver lenta, o Python não vai estar na tela. A consulta vai.

AS_QUATRO_OPERACOES.SQL

sql

```

1  INSERT INTO produto (nome, preco, estoque)
2  VALUES ('Tomate italiano', 8.90, 120);
3
4  SELECT id, nome, preco FROM produto
5  WHERE estoque > 0
6  ORDER BY nome
7  LIMIT 20;
8
9  UPDATE produto SET preco = 9.50 WHERE id = 1;
10
11 DELETE FROM produto WHERE id = 1;

```

Quatro comandos, os mesmos quatro verbos do CRUD. A correspondência é direta e é o motivo de “CRUD” e “SQL” aparecerem sempre na mesma frase.

⚠ ERRO COMUM

UPDATE e DELETE **sem** WHERE atingem a tabela inteira. Não há confirmação, não há desfazer, e o banco não acha isso estranho — é um comando válido. O hábito que salva: escreva primeiro o WHERE, depois volte e escreva o começo do comando.

20.5 Índice: por que a busca fica lenta

```
SELECT * FROM produto WHERE nome = 'Tomate italiano';
```

Sem índice, o banco lê **todas** as linhas e compara uma a uma. Com dez linhas, é instantâneo. Com dez milhões, é o suporte ligando.

```
CREATE INDEX idx_produto_nome ON produto (nome);
```

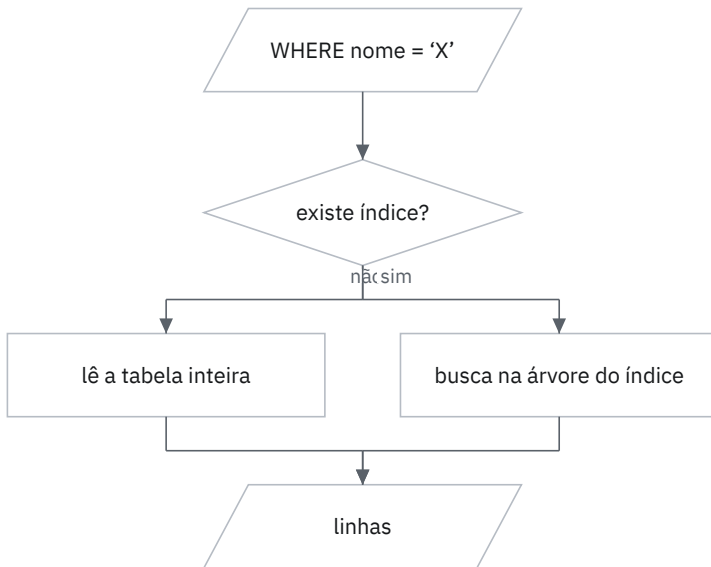


FIGURA 20.1 Sem índice, o banco lê tudo. Com índice, ele consulta um atalho ordenado.

O índice não é grátis: ele ocupa espaço e torna toda escrita um pouco mais lenta, porque precisa ser atualizado junto. A regra: indexe o que aparece em `WHERE`, em `JOIN` e em `ORDER BY` — e nada além disso, sem medir.

A chave primária ganha índice automaticamente. Chave estrangeira, na maioria dos bancos, **não** — e essa ausência é a causa mais comum de listagem lenta em sistema pequeno.

20.6 Transação: tudo ou nada

Aqui está a garantia que nenhum arquivo JSON oferece.

TRANSACAO.SQLsql

```
1 CREATE TABLE movimento (  
2     id          BIGSERIAL PRIMARY KEY,  
3     produto_id  BIGINT NOT NULL REFERENCES produto(id),  
4     quantidade  INTEGER NOT NULL,  
5     criado_em   TIMESTAMPTZ NOT NULL DEFAULT NOW()  
6 );  
7  
8 BEGIN;  
9  
10 UPDATE produto SET estoque = estoque - 10 WHERE id = 1;  
11 INSERT INTO movimento (produto_id, quantidade) VALUES (1, -10);  
12  
13 COMMIT;
```

Entre `BEGIN` e `COMMIT`, ou as duas linhas acontecem, ou nenhuma. Se o servidor cair no meio, o banco desfaz tudo ao voltar. Se alguém consultar o estoque durante a transação, vê o valor antigo — não vê o estado pela metade.

As quatro letras de ACID descrevem essa promessa:

LETRA	PROMESSA
Atomicidade	tudo ou nada
Consistência	as regras da tabela continuam valendo
Isolamento	uma transação não enxerga o meio da outra
Durabilidade	depois do <code>COMMIT</code> , sobrevive à queda de energia

TABELA 20.2 São quatro promessas caras de implementar e é exatamente por elas que se usa um banco de dados.

● **O ESSENCIAL**

A transação é a unidade de trabalho do negócio, não a unidade técnica. “Dar baixa no estoque e registrar o movimento” é uma operação só do ponto de vista da cooperativa; o fato de serem dois comandos SQL é detalhe. Essa fronteira é o que define, no capítulo 23, onde a transação começa e termina.

20.7 Conectando do Python

```
$ pip install "psycopg[binary]"
```

CONEXAO_CRUA.PY

python

```

1  import psycopg
2
3  url = "postgresql://postgres:senha@localhost/catalogo"
4
5  with psycopg.connect(url) as conn:
6      with conn.cursor() as cur:
7          cur.execute(
8              "INSERT INTO produto (nome, preco) VALUES (%s, %s)",
9              ("Cenoura", "5.25"),
10             )
11          cur.execute("SELECT id, nome, preco FROM produto")
12          for linha in cur.fetchall():
13              print(linha)

```

Os dois `with` do capítulo 14 aparecem aqui com um papel a mais: o `with` da conexão faz `COMMIT` ao sair sem erro e `ROLLBACK` se uma exceção escapar.

▲ ATENÇÃO

Repare no `%s` e na tupla separada. **Nunca** monte SQL com f-string:

```
cur.execute(f"SELECT * FROM produto WHERE nome = '{nome}'")
```

Se `nome` vier de uma requisição e contiver `' ; DROP TABLE produto; --`, o banco executa o que foi mandado. Isso se chama injeção de SQL, tem quase trinta anos, e continua sendo uma das falhas mais exploradas do mundo. Com `%s`, o valor viaja separado do comando e nunca é interpretado como comando.

ENQUANTO ISSO, NA EMPRESA A planilha e o banco, lado a lado

— E se a gente perder tudo? — perguntou Dona Neuza.

— A gente faz cópia de segurança todo dia — disse Bia.

— A planilha eu abro e vejo. Esse negócio aí eu não vejo.

Bia pensou um instante e abriu o `psql`. Digitou `SELECT * FROM produto`; e virou a tela. As linhas apareceram em colunas, com cabeçalho, alinhadas.

— É uma planilha — disse Dona Neuza.

— É uma planilha que sabe dizer não. — Bia digitou de novo, agora com um preço negativo. O banco recusou, com uma linha vermelha e o nome da regra violada. — A sua abre e aceita. Essa aqui abre, olha, e não deixa.

Dona Neuza leu a mensagem de erro com atenção, de óculos na ponta do nariz.

— Ela recusa também se eu mandar? — perguntou.

— Recusa a senhora, recusa a mim e recusa o sistema.

RESUMO

- Arquivo não resolve concorrência, busca, atomicidade nem escrita parcial.
- `NUMERIC` para dinheiro, `TIMESTAMP` para instante, `NOT NULL` por padrão.
- Ler SQL é obrigatório mesmo usando ORM; escrever, quase nunca.
- `UPDATE` e `DELETE` sem `WHERE` atingem tudo.
- Índice acelera leitura e custa em escrita; chave estrangeira não ganha um sozinha.
- Transação é a unidade de trabalho do negócio, e ACID é o que ela promete.
- Valor nunca entra em SQL por `f-string`.

■ O PROJETO AGORA

O banco existe, a tabela existe e o Python conversa com ele. O código continua feio — SQL cru dentro de rota seria pior que a lista na memória — e o capítulo 21 resolve isso.

- 1 Crie a tabela `produtor` com `id`, `nome` (obrigatório), `cidade` (obrigatório), `email` (único) e `criado_em` com valor padrão.
- 2 Escreva a consulta que devolve os cinco produtos mais caros com estoque disponível, mostrando nome e preço.
- 3 A cooperativa quer registrar a venda de 10 caixas: baixar o estoque do produto e gravar um movimento. Escreva a transação e depois explique o que acontece, em cada uma das duas linhas, se o servidor cair exatamente entre elas.

CAPÍTULO

21

SQLAlchemy

O ORM não esconde o banco. Ele traduz — e quem não sabe a língua de origem não percebe quando a tradução ficou ruim.

AO FIM DESTES CAPÍTULO

Mapear uma tabela para uma classe, abrir e fechar sessões corretamente, escrever consultas com `select()` e entender a diferença entre `flush` e `commit`.

**Capítulo disponível na
versão completa**

CAPÍTULO

22

Repositório

A camada que conhece o banco — e a única do projeto que tem esse direito.

AO FIM DESTE CAPÍTULO

Isolar o acesso a dados numa camada própria, definir o contrato com `Protocol`, e entender por que o repositório não confirma transação nem decide regra.

**Capítulo disponível na
versão completa**

CAPÍTULO

23

Serviço

A camada onde mora a cooperativa — e não o HTTP, nem o SQL.

AO FIM DESTES CAPÍTULOS

Concentrar as regras de negócio numa camada própria, definir a fronteira da transação, levantar exceções de domínio e escrever código que pode ser testado sem servidor e sem banco.

**Capítulo disponível na
versão completa**

CAPÍTULO

24

O CRUD completo

*As quatro camadas ligadas, as cinco rotas no ar,
e o primeiro dia em que o dado sobrevive ao
reinício.*

AO FIM DESTES CAPÍTULOS

Ligar router, service, repository e modelo com o sistema de dependências do FastAPI, e ter o CRUD inteiro funcionando sobre PostgreSQL.

**Capítulo disponível na
versão completa**

PARTE 5

API de verdade

O que separa um CRUD de exercício de uma API que outra pessoa usa: validação, erro honesto, paginação e busca.

NESTA PARTE

- 25 Esquemas
- 26 Validação
- 27 Tratamento de erros
- 28 Paginação
- 29 Filtros e buscas

CAPÍTULO

25

Esquemas

O esquema é o contrato público. A tabela é assunto interno — e confundir os dois é publicar o seu banco.

AO FIM DESTA CAPÍTULO

Desenhar os esquemas de entrada, saída e alteração parcial; distinguir campo ausente de campo nulo; e manter o contrato estável enquanto o modelo muda.

**Capítulo disponível na
versão completa**

CAPÍTULO

26

Validação

Três camadas validam a mesma coisa, e nenhuma das três é redundante.

AO FIM DESTA CAPÍTULO

Distribuir a validação entre esquema, serviço e banco; escrever validadores que normalizam; e produzir mensagens de erro que quem consome a API consegue usar.

**Capítulo disponível na
versão completa**

CAPÍTULO

27

Tratamento de erros

A resposta de erro é a parte da API que mais gente lê e menos gente projeta.

AO FIM DESTE CAPÍTULO

Substituir os `try` espalhados por tratadores globais, padronizar o corpo de erro, registrar o que interessa e nunca vaziar detalhe interno.

**Capítulo disponível na
versão completa**

CAPÍTULO

28

Paginação

Toda listagem sem limite é uma promessa que o servidor não consegue cumprir para sempre.

AO FIM DESTES CAPÍTULOS

Paginar com `limit/offset`, devolver metadados úteis, garantir ordenação estável e saber quando trocar por paginação por cursor.

**Capítulo disponível na
versão completa**

CAPÍTULO

29

Filtros e buscas

Toda busca começa com uma comparação injusta: o usuário digita sem acento e o banco guarda com.

AO FIM DESTE CAPÍTULO

Montar consultas com filtros opcionais sem escadas de `if`, ordenar por campo escolhido pelo cliente com segurança, e escolher entre `LIKE`, busca sem acento e busca textual de verdade.

**Capítulo disponível na
versão completa**

PARTE 6

O Python que ninguém vê no tutorial

Relacionamentos, migrações, injeção de dependência e async — onde o framework fica poderoso e perigoso na mesma medida.

NESTA PARTE

- 30 Relacionamentos
- 31 Migrações
- 32 Injeção de dependência
- 33 Assíncrono

CAPÍTULO

30

Relacionamentos

O ORM faz parecer que o objeto já tem tudo dentro. A conta chega uma consulta por vez.

AO FIM DESTE CAPÍTULO

Modelar um-para-muitos e muitos-para-muitos, escolher a estratégia de carregamento, e reconhecer o problema N+1 antes de ele chegar à produção.

**Capítulo disponível na
versão completa**

31

Migrações

O banco de produção não pode ser recriado. Ele só pode ser transformado, um passo por vez, na mesma ordem em todo lugar.

AO FIM DESTES CAPÍTULOS

Trocar `create_all` pelo Alembic, gerar e revisar migrações, escrever migração de dados, e planejar uma alteração que não derruba a aplicação.

**Capítulo disponível na
versão completa**

CAPÍTULO

32

Injeção de dependência

O framework monta o grafo de objetos a cada requisição — e você o descreve com anotações de tipo.

AO FIM DESTES CAPÍTULOS

Entender decoradores por dentro, escrever dependências com e sem `yield`, encadeá-las, e substituí-las em teste sem tocar no código de produção.

**Capítulo disponível na
versão completa**

CAPÍTULO

33

Assíncrono

Concorrência não é velocidade. É deixar de ficar parado esperando.

AO FIM DESTE CAPÍTULO

Entender o laço de eventos, escrever rotas `async` sem travá-lo, decidir entre rota síncrona e assíncrona, e saber o que muda no acesso ao banco.

**Capítulo disponível na
versão completa**

PARTE 7

Segurança

Autenticação, autorização e um token que diz quem você é sem que o servidor precise lembrar.

NESTA PARTE

- 34 Autenticação
- 35 JWT
- 36 Usuários e permissões

CAPÍTULO

34

Autenticação

Guardar senha é a única parte do sistema em que improvisar é indefensável.

AO FIM DESTE CAPÍTULO

Guardar senhas com hash lento, escrever o cadastro e o login, entender a diferença entre sessão e token, e reconhecer os erros clássicos de quem implementa autenticação pela primeira vez.

**Capítulo disponível na
versão completa**

CAPÍTULO

35

JWT

Um crachá que o servidor emite, assina e não guarda — e é o “não guarda” que custa caro.

AO FIM DESTE CAPÍTULO

Emitir e verificar tokens assinados, escolher o conteúdo e a validade, implementar renovação, e entender por que revogar um JWT é difícil.

**Capítulo disponível na
versão completa**

CAPÍTULO

36

Usuários e permissões

Saber quem é a pessoa responde metade da pergunta. A outra metade é o que ela pode fazer com o dado que é de outro.

AO FIM DESTE CAPÍTULO

Modelar papéis, proteger rotas por papel, verificar propriedade do recurso e escolher a camada certa para cada tipo de autorização.

**Capítulo disponível na
versão completa**

PARTE 8

Testes

*A rede que permite mexer no código sem medo.
Unidade, integração e um banco de verdade
dentro do teste.*

NESTA PARTE

- 37 Testando Python
- 38 Testando serviços
- 39 Testando a API
- 40 Testando o banco

CAPÍTULO

37

Testando Python

O teste não prova que o código está certo. Ele prova que aquilo que já funcionava continua funcionando.

AO FIM DESTE CAPÍTULO

Escrever testes com pytest, usar fixtures e parametrização, nomear testes que explicam a falha, e decidir o que merece teste.

**Capítulo disponível na
versão completa**

CAPÍTULO

38

Testando serviços

As regras da cooperativa não precisam de banco para serem verdadeiras — nem para serem testadas.

AO FIM DESTE CAPÍTULO

Testar a camada de serviço com dublês, escolher entre falso e simulacro, e escrever testes de regra que rodam em milissegundos.

**Capítulo disponível na
versão completa**

CAPÍTULO

39

Testando a API

O teste de API não verifica a regra. Ele verifica o contrato: status, corpo e cabeçalho.

AO FIM DESTE CAPÍTULO

Escrever testes de integração com `TestClient`, substituir dependências, testar rotas protegidas e evitar que o conjunto vire uma suíte lenta e frágil.

**Capítulo disponível na
versão completa**

CAPÍTULO

40

Testando o banco

Um teste verde em outro banco é uma opinião sobre outro sistema.

AO FIM DESTES CAPÍTULOS

Rodar testes contra um PostgreSQL de verdade, isolar cada teste por transação, subir o banco com contêiner e testar o que só o banco garante.

**Capítulo disponível na
versão completa**

PARTE 9

Documentação e ferramentas

Uma API que ninguém sabe usar não existe. E um projeto sem histórico não tem volta.

NESTA PARTE

41 OpenAPI

42 Git

CAPÍTULO

41

OpenAPI

Documentação que não nasce do código começa a mentir no dia seguinte.

AO FIM DESTE CAPÍTULO

Enriquecer a documentação gerada, descrever respostas de erro, versionar a API e usar o esquema para gerar clientes em outras linguagens.

**Capítulo disponível na
versão completa**

CAPÍTULO

42

Git

Um projeto sem histórico não tem volta — e a volta é a única coisa que se quer quando algo dá errado.

AO FIM DESTES CAPÍTULOS

Versionar o projeto com um histórico legível, trabalhar em ramos, proteger o repositório de segredos e automatizar a conferência antes do commit.

O projeto tem quarenta arquivos, quatro camadas, migrações e testes. Ele ainda mora numa pasta sem memória: qualquer alteração sobrescreve a anterior, e “como estava na semana passada” é uma pergunta sem resposta.

42.1 O começo

```
$ git init
$ git add .
$ git commit -m "primeiro commit: catálogo da Cooperativa Sabiá"
```

Antes do `git add .`, porém, existe um arquivo que precisa existir.

42.2 `.gitignore`, antes de tudo

.GITIGNORE	text
1 # ambiente	
2 .venv/	
3 __pycache__/	
4 *.py[cod]	
5	
6 # configuração local — NUNCA versionar	
7 .env	
8 *.pem	
9 *.key	
10	
11 # ferramentas	
12 .pytest_cache/	
13 .coverage	
14 htmlcov/	
15 .mypy_cache/	
16 .ruff_cache/	
17	
18 # editor e sistema	
19 .idea/	
20 .vscode/	
21 .DS_Store	

▲ ATENÇÃO

`.env` no `.gitignore` é a linha mais importante aqui. Um segredo que entrou no histórico **não sai** apagando o arquivo: ele continua em todos os commits anteriores, em todos os clones, em todos os forks e na cópia que a plataforma manteve. A resposta a uma chave vazada não é apagar o commit — é **trocar a chave**, imediatamente, antes de qualquer outra coisa.

O `.env.example` do capítulo 17 entra no repositório justamente porque não tem valor real: ele documenta quais variáveis existem sem revelar nenhuma.

42.3 Commits que contam uma história

```
$ git add app/services/produto.py tests/test_servico_produto.py
$ git commit
```

```
1  Recusa venda acima do estoque disponível
2
3  A rotina de ajuste em lote subtraía do estoque sem conferir,
4  e quatro produtos ficaram negativos no fechamento de novembro.
5
6  A verificação passa para o serviço, junto com um CHECK no
7  banco — o serviço dá a mensagem, o banco dá a garantia.
8
9  Refs #214
```

A primeira linha é um **resumo no imperativo**, até cinquenta caracteres, sem ponto final. Ela aparece sozinha em `git log --oneline`, na lista de commits da plataforma e na busca — e é a única linha que muita gente vai ler.

O corpo explica **por quê**. O “o quê” está no diff; o “por quê” só existe na cabeça de quem escreveu, e some em duas semanas.

↳ ERRO COMUM

`git commit -m "ajustes"` é o commit que não ajuda ninguém. Seis meses depois, procurando quando um comportamento mudou, você vai encontrar quinze deles em sequência e vai precisar abrir os quinze. O custo de escrever uma linha boa é de trinta segundos; o de não escrever é pago por outra pessoa, com juros.

Um bom tamanho de commit é **uma mudança com um motivo**. Consertar um defeito e reformatar o arquivo são dois motivos, e dois commits — senão a correção de três linhas fica escondida numa mudança de trezentas.

42.4 Ramos

```
$ git switch -c 214-estoque-negativo
$ ...trabalho...
$ git switch main
$ git merge 214-estoque-negativo
$ git branch -d 214-estoque-negativo
```

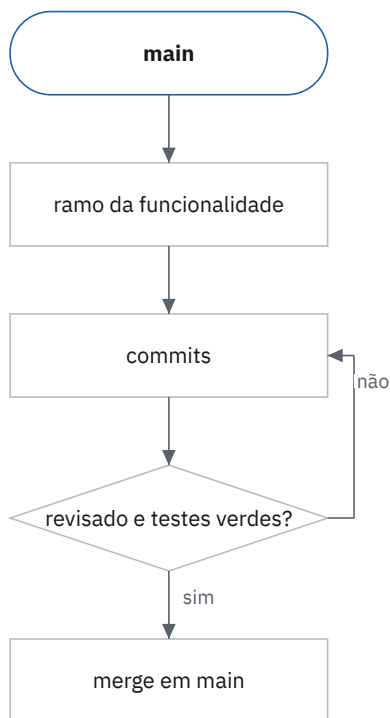


FIGURA 42.1 Um ramo por funcionalidade, revisão antes de entrar.

A `main` é o que pode ir para produção a qualquer momento. Tudo o mais acontece em ramo.

42.5 Conflito

```
<<<<<<< HEAD
    limite = max(1, min(limite, 100))
=====
    limite = min(limite, 200)
>>>>>> 214-estoque-negativo
```

Entre <<<<<< e ===== está a versão de quem estava lá; entre ===== e >>>>>>, a sua. Resolver é escolher, combinar ou reescrever — e **apagar as três linhas de marcação**.

⚠ ERRO COMUM

Deixar um <<<<<< no arquivo produz um `SyntaxError` que costuma aparecer só em produção, porque ninguém rodou aquele módulo depois do merge. Uma busca por <<<<<< em todo o projeto, antes de confirmar, leva um segundo — e o gancho de pré-commit da próxima seção faz isso por você.

42.6 Ganchos de pré-commit

```
$ pip install pre-commit
```

.PRE-COMMIT-CONFIG.YAML

text

```
1  repos:
2    - repo: https://github.com/astral-sh/ruff-pre-commit
3      rev: v0.8.4
4      hooks:
5        - id: ruff
6          args: [--fix]
7        - id: ruff-format
8
9    - repo: https://github.com/pre-commit/pre-commit-hooks
10      rev: v5.0.0
11      hooks:
12        - id: check-merge-conflict
13        - id: check-added-large-files
14        - id: detect-private-key
15        - id: end-of-file-fixer
```

```
$ pre-commit install
```

A partir daí, todo `git commit` roda essas conferências e recusa o commit se alguma falhar. O `detect-private-key` e O `check-merge-conflict` sozinhos já pagam a instalação.

● O ESSENCIAL

Formatação automática encerra uma categoria inteira de discussão. Com o `ruff-format` no gancho, ninguém revisa espaçamento, ninguém debate aspas simples contra duplas, e nenhum diff vem poluído por reformatação. A ferramenta decide, e o time discute o que importa.

42.7 Integração contínua

.GITHUB/WORKFLOWS/CI.YML

text

```
1  name: CI
2  on: [push, pull_request]
3
4  jobs:
5    testes:
6      runs-on: ubuntu-latest
7      services:
8        postgres:
9          image: postgres:16
10         env:
11           POSTGRES_USER: catalogo
12           POSTGRES_PASSWORD: senha
13           POSTGRES_DB: catalogo_test
14         ports: ["5432:5432"]
15     steps:
16       - uses: actions/checkout@v4
17       - uses: actions/setup-python@v5
18         with:
19           python-version: "3.12"
20           cache: pip
21       - run: pip install -r requirements.txt
22       - run: ruff check app tests
23       - run: mypy app
24       - run: pytest -q
```

Os testes do capítulo 40 rodam a cada envio, num PostgreSQL de verdade, na máquina de outra pessoa. É o que transforma “passa na minha máquina” em uma afirmação verificável.

ENQUANTO ISSO, NA EMPRESA Funciona na minha máquina

A esteira ficou vermelha por três dias seguidos, sempre no mesmo teste, e na máquina de todo mundo ele passava.

Rafa acabou encontrando: o teste dependia da ordem em que o banco devolvia os produtos. Localmente, com quatro registros, o PostgreSQL devolvia na ordem de inserção. Na esteira, com o banco recém-criado e estatísticas diferentes, ele escolhia outro plano.

— Então o teste estava errado.

— O teste estava **incompleto** — disse Bia. — Faltava o `ORDER BY`. E o código que ele testava também.

Era a mesma ausência do capítulo da paginação, escondida num lugar onde ninguém tinha olhado.

42.8 Comandos que salvam

COMANDO	PARA QUÊ
<code>git log --oneline -20</code>	os últimos vinte commits
<code>git log -p arquivo.py</code>	o histórico de um arquivo
<code>git blame arquivo.py</code>	quem escreveu cada linha, e quando
<code>git diff</code>	o que mudou e ainda não foi adicionado
<code>git restore arquivo.py</code>	desfazer alteração não adicionada
<code>git revert <hash></code>	desfazer um commit criando outro

TABELA 42.1 `git blame` não serve para achar culpado. Serve para achar o commit — e, nele, a explicação de por que aquela linha existe.

▲ ATENÇÃO

`git push --force` reescreve o histórico do servidor. Em ramo pessoal, é aceitável. Em ramo compartilhado, apaga o trabalho de quem tinha baixado. Se precisar mesmo, `--force-with-lease` recusa a operação caso alguém tenha enviado algo desde a sua última atualização.

RESUMO

- `.gitignore` antes do primeiro `git add .env` nunca entra.
- Segredo que entrou no histórico se resolve trocando o segredo.
- Resumo no imperativo, corpo explicando o porquê.
- Um commit é uma mudança com um motivo.
- Ganchos de pré-commit pegam conflito, chave privada e formatação.
- A esteira transforma “funciona na minha máquina” em algo verificável.
- `--force` em ramo compartilhado apaga o trabalho dos outros.

- 1 Escreva o `.gitignore` do projeto e faça o primeiro commit, conferindo com `git status` que o `.env` não entrou.
- 2 Escreva a mensagem de commit para a correção de um `PATCH` que apagava campos não enviados.
- 3 Você descobre que uma chave de API de um fornecedor foi commitada há três meses, no repositório privado da empresa, e já passou por dois forks internos. Descreva o que fazer, em ordem.

PARTE 10

Projeto final

Agora sem mastigar. Você recebe os requisitos e constrói a loja inteira — e depois a coloca no ar.

NESTA PARTE

CAPÍTULO

43

Projeto final

Agora sem mastigar: os requisitos, o prazo e o dia em que alguém de fora usa o que você fez.

AO FIM DESTE CAPÍTULO

Construir sozinho um módulo completo a partir de requisitos, empacotar a aplicação em contêiner e colocá-la no ar com o que uma API de verdade exige.

**Capítulo disponível na
versão completa**

Respostas

O que vamos construir

- 1 Ele pediu existência e preço de um produto. Ela responde com um número, ou com “acabou”.

Ele não precisou saber em qual aba está o preço, qual célula não pode rodar, nem se o número saiu do notebook ou da planilha. Isso é o contrato. O endereço e o verbo, quando existirem, são só essa conversa escrita num formato que um programa consegue repetir.

2

..

```
POST    /produtos    cadastra
GET     /produtos    lista
GET     /produtos/12    vê um
PUT     /produtos/12    troca
DELETE  /produtos/12    tira
```

Os dois `GET` são o ponto: o mesmo verbo, num endereço e noutro, significa coisas diferentes. O verbo diz a intenção. O endereço diz a coisa.

- 3 Não. `baratos` não é um produto — é um filtro sobre os produtos. No dia em que aparecerem `/produtos/caros`, `/produtos/em-falta` e `/produtos/organicos`, serão quatro endereços que não se combinam: não há como pedir “barato e orgânico”.

A forma que combina é `GET /produtos?preco_max=10`. O endereço diz *o que*. O que vem depois do `?` diz *quais*.

O que é o FastAPI

1

```
@app.get("/versao")
def versao():
    return {"versao": "0.1.0"}
```

2

..

```
1 @app.get("/ping")
2 def ping_ler():
3     return {"metodo": "GET"}
4
5
6 @app.post("/ping")
7 def ping_criar():
8     return {"metodo": "POST"}
```

A rota, em HTTP, é o par **verbo + caminho**, não o caminho sozinho. É por isso que o CRUD inteiro cabe em dois caminhos: `/produtos` e `/produtos/{id}`, com verbos diferentes em cada um.

- 3** Nenhum dos dois resolve os três problemas bem sozinho, e essa é a resposta honesta. O Django entrega o painel administrativo pronto — que no FastAPI seria um projeto inteiro — e resolve relatório com templates maduros. O FastAPI entrega a API pública com documentação gerada e validação tipada, que no Django exigiria o Django REST Framework e bastante configuração.

A recomendação que eu escreveria: **Django para o painel e os relatórios, FastAPI para a API pública, com o banco compartilhado**. E registraria por escrito o preço dessa escolha — dois projetos, duas implantações, duas pilhas de dependência — para que a decisão seja revista se o painel encolher ou a API crescer. Escolher uma ferramenta só, aceitando o atrito num dos lados, é defensável; o que não é defensável é escolher sem nomear o atrito.

Banco de dados

1

```
1 CREATE TABLE produtor (
2     id          BIGSERIAL PRIMARY KEY,
3     nome        VARCHAR(100) NOT NULL,
4     cidade      VARCHAR(100) NOT NULL,
5     email       VARCHAR(160) NOT NULL UNIQUE,
6     criado_em   TIMESTAMPTZ NOT NULL DEFAULT NOW()
7 );
```

`UNIQUE` cria um índice sozinho — é assim que o banco consegue conferir a unicidade sem varrer a tabela a cada inserção.

2

```
SELECT nome, preco
FROM produto
WHERE estoque > 0 AND ativo = TRUE
ORDER BY preco DESC
LIMIT 5;
```

A ordem em que o banco trabalha não é a ordem em que você escreve: ele filtra com o `WHERE`, depois ordena, e só então corta com o `LIMIT`. Saber isso é o que explica por que `LIMIT 5` numa tabela grande ainda pode ser lento — o `ORDER BY` precisou tocar todas as linhas que passaram no filtro.

3

...

```

1 BEGIN;
2
3 UPDATE produto
4     SET estoque = estoque - 10
5     WHERE id = 1 AND estoque >= 10;
6
7 INSERT INTO movimento (produto_id, quantidade, tipo)
8 VALUES (1, -10, 'venda');
9
10 COMMIT;

```

Se o servidor cair entre as duas linhas, **nada aconteceu**. O `UPDATE` ficou registrado apenas no log de transação, sem `COMMIT` ao reiniciar, o banco desfaz. Ninguém vê estoque baixado sem movimento correspondente, nem por um instante.

Repare no `AND estoque >= 10` dentro do `WHERE`. Ele é a proteção contra estoque negativo feita **no mesmo comando** que altera — e não num `SELECT` anterior. Entre um `SELECT` de conferência e um `UPDATE` separado existe uma janela em que outra transação pode ter vendido as mesmas caixas. Condição e alteração juntas no mesmo comando fecham essa janela, porque o banco avalia a linha travada.

O que ainda falta é saber se a venda aconteceu: se ninguém tinha as 10 caixas, o `UPDATE` afeta zero linhas e o `INSERT` grava um movimento que não existiu. Conferir o número de linhas afetadas, e desfazer quando for zero, é a parte que o capítulo 23 vai escrever em Python.

Git

1

```

$ git init
$ printf '.venv/\n__pycache__/\n.env\n' > .gitignore
$ git add .
$ git status --short

```

Se `.env` aparecer na lista, ele já estava rastreado — o `.gitignore` só vale para arquivos ainda não versionados. O conserto é `git rm --cached .env` antes do commit.

2

..

```

1  Corrige PATCH que apagava campos não enviados
2
3  model_dump() sem exclude_unset devolvia None para todo campo
4  ausente, e o serviço gravava esses None por cima dos valores
5  existentes. Um PATCH só com o preço apagava o nome do
   produto.
6
7  Passa a usar exclude_unset=True e acrescenta teste com um
8  único campo no corpo.
9
10 Refs #241

```

A segunda linha em branco entre resumo e corpo não é estética: o Git usa exatamente essa separação para distinguir os dois.

3

...

Agora, antes de qualquer coisa relacionada ao Git. Enquanto a chave for válida, tudo o mais é secundário — e limpar o histórico sem revogar é gastar horas numa tarefa que não resolve o problema.

2. Emitir uma chave nova

e colocá-la onde ela deveria estar desde o início: variável de ambiente, ou um cofre de segredos.

3. Verificar o uso.

O fornecedor tem log de chamadas: veja se houve uso fora dos seus endereços e horários habituais. Essa é a diferença entre “chave exposta” e “chave usada”, e ela muda o que precisa ser comunicado.

4. Comunicar.

Segurança da empresa e o fornecedor. Um repositório privado com dois forks internos não é um incidente público, e continua sendo um incidente.

5. Só então, o histórico.

`git filter-repo` ou o BFG reescrevem os commits antigos, e isso exige que **todo mundo** apague os clones e baixe de novo — inclusive os forks. É trabalhoso, quebra todas as referências de commit e, com a chave já revogada, o benefício é modesto: evitar que o valor antigo apareça numa auditoria futura. Em repositório privado com poucos clones, muita gente decide, com razão, não reescrever.

6. Impedir a repetição.

`detect-secrets` ou `gitLeaks` no gancho de pré-commit e na esteira. Esse passo é o único que muda alguma coisa para o próximo

trimestre — os cinco anteriores tratam deste incidente; só o sexto trata do próximo.

Glossário

API *Application Programming Interface*, interface de programação. É um contrato entre dois programas: um sabe fazer alguma coisa, o outro precisa que ela seja feita, e existe uma forma acordada de pedir e de responder.

GIL *Global Interpreter Lock*: uma trava que, no Python padrão, deixa apenas uma thread executar código Python de cada vez. Existe para o gerenciador de memória não se perder. Para a maior parte das APIs, o tempo está no banco e na rede, não nessa trava. Ela importa em conta pesada, em paralelo, dentro do processo. Desde o 3.13 existe um modo sem ela, e esse modo não é o que se instala por padrão. Quem sofre com o GIL sabe que sofre. Quem repete a sigla numa reunião, em geral, não mediu.

Conheça as demais obras dessa coleção



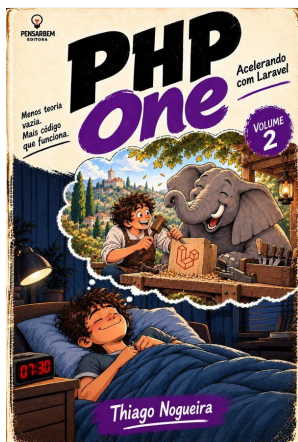
Java One

Do primeiro println à API REST em produção



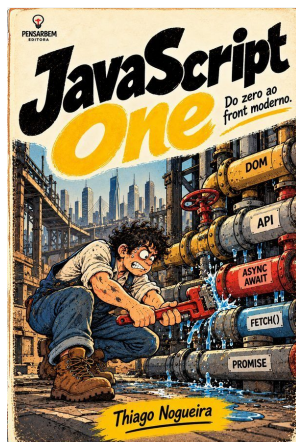
PHP One

Volume 1 · Fundamentos do PHP



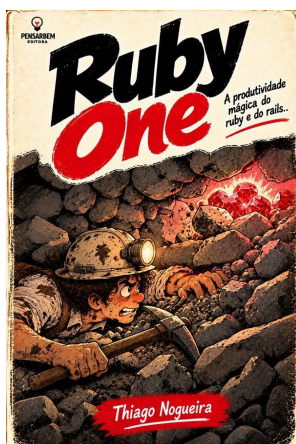
PHP One

Volume 2 · Acelerando com Laravel



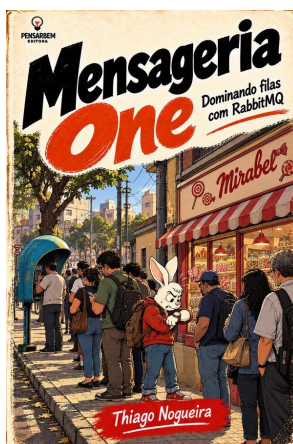
JavaScript One

Do primeiro console.log ao painel em produção



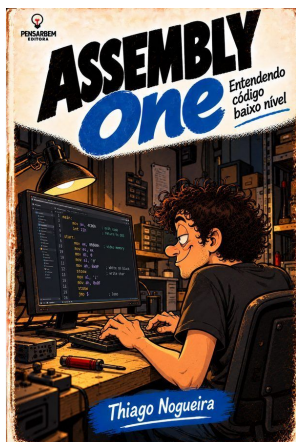
Ruby One

Do primeiro puts ao contrato em produção



Mensageria One

RabbitMQ do primeiro publish à Black Friday sem perder pedido



Assembly One

Do zero à máquina MARIE

Visite a editora Pensar Bem →

pensarbem.com.br