Python para todos: um guia acessível

Eduardo Bulsoni

27 de julho de 2025

Sumário

Capítulo 1

Primeiros passos com python!

1.1 Por que python?

Python é uma linguagem simples, poderosa e extremamente versátil. Ela pode ser usada em diversas áreas: ciência de dados, desenvolvimento de jogos, criação de sites, aplicativos, aprendizado de máquina, automações, administração de sistemas e muito mais. Se o foco do seu projeto não for performance extrema, como é o caso de linguagens como C, é bem provável que você consiga resolver com Python.

Em outras palavras, se você está começando sua jornada no mundo da programação, Python é como um bote salva-vidas que pode te ajudar a atravessar qualquer mar de códigos! (risos)

1.2 O que é programação?

Programação é o ato de escrever instruções em forma de texto (código) que o computador deve seguir. Essas instruções precisam ser simples, passo a passo, e, o mais importante, sem ambiguidades!

Você já ouviu aquela piada do programador? A esposa diz: "Amor, quero que você vá ao mercado e traga 2 ovos. Se tiver leite, traga 6."O homem vai até o mercado, encontra leite, mas volta com 6 ovos e nenhum leite. Pode parecer engraçado, mas eu te pergunto: a instrução estava clara?

1.2.1 Uma breve história da ciência da computação: como chegamos até aqui

Embora o autor deste livro não seja exatamente um entusiasta de histórias antigas, é impossível ignorar como a ciência da computação percorreu um longo caminho para chegar ao que temos hoje.

Nos primórdios, programar um computador era uma tarefa bem diferente do que é atualmente. Os programas eram escritos em cartões perfurados, que precisavam ser inseridos em grandes máquinas para processamento. Cada pequeno erro, como trocar a ordem de um cartão, poderia comprometer todo o programa. Além disso, os computadores eram imensos, ocupando salas inteiras, e a ideia de ter um em casa era impensável.

Com o tempo, evoluímos para os chamados terminais burros, dispositivos conectados a servidores centrais. Embora esses terminais permitissem aos usuários escrever comandos, o trabalho pesado era feito por computadores maiores e mais poderosos. Ainda assim, a experiência deixava a desejar: havia um atraso considerável entre o pressionar de uma tecla e a resposta do servidor.

Hoje, a realidade é completamente diferente. Graças à evolução da computação, você pode escrever centenas ou milhares de linhas de código e obter resultados em segundos ou minutos. O que antes exigia salas inteiras e equipamentos especializados agora pode ser feito em um laptop ou até mesmo em um smartphone.

E é impossível falar da história da computação sem mencionar Alan Turing: matemático brilhante, pioneiro da ciência da computação, e uma figura histórica cujas ideias revolucionaram o campo. Seu trabalho nos ajuda a entender como chegamos a um ponto em que programar se tornou acessível a todos. Obrigado, Turing!

1.3 Ferramentas e como instalá-las

Bom, a partir daqui eu assumo que você tenha acesso à algum computador (pode ser até um chromebook ou coisa que o valha), de todo modo, essas serão nossas principais ferramentas:

- Visual Studio Code
- Python
- Pycharm

1.3.1 Instalando o VSCode

A instalação do VSCode é bem simples. Ao baixar o arquivo, basta seguir os passos do instalador. Durante o processo, você verá algumas opções, como a de adicionar o VSCode ao contexto da pasta atual (uma opção muito útil, pois facilita a abertura de projetos diretamente do explorador de arquivos). Não é obrigatório, mas pode ser interessante para facilitar sua vida mais tarde.

1.3.2 Instalando o Python

Baixe o Python; quando for instalar, se você tiver permissões de administrador, você pode optar por instalar o Python para todos os usuários do computador.

Importante

Não se esqueça de marcar a opção "Add Python to PATH"(Adicionar Python ao PATH) durante a instalação. Isso é essencial para que você possa rodar o Python diretamente do terminal (sem precisar navegar até a pasta onde o Python está instalado).

1.3.3 Instalando o PyCharm

Se você optar por usar o PyCharm, cuidado! Ao entrar no site da jetbrains para fazer o download, você verá duas opções: PyCharm Professional (pago) e PyCharm Community Edition (GRATUITO). Role a página até encontrar a versão Community Edition e baixe-a. Essa versão é perfeita para quem está começando.

1.4 Meu primeiro programa

Agora que já sabemos o que é programação, estamos prontos para escrever nossa primeira linha de código! Digite o seguinte comando no seu editor de texto:

```
1 print("Ola Mundo!")
```

Código 1.1: Olá mundo em python

Salve o arquivo com o nome "ola_mundo.py"(com a extensão .py, que é fundamental para que o Python reconheça o arquivo como um script Python). Agora, você pode rodar o seu código de duas maneiras:

1. No editor de texto: Normalmente, você encontrará um botão de "play"ou "executar"para rodar o script diretamente no editor.
2. No terminal: Se preferir usar o terminal, abra a pasta onde o arquivo está salvo e digite o seguinte comando: python ola_mundo.py

Importante

A extensão .py é essencial para que o arquivo seja identificado como um script Python.
Além disso, a convenção do Python para nomes de arquivos é utilizar o snakecase — ou seja, todas as palavras em minúsculas, separadas por underscores (sublinhados).
Então, lembre-se: ao nomear seus arquivos, use essa convenção para manter o código organizado e legível.

O que faz o nosso primeiro *"Olá, Mundo"*?

O comando que escrevemos, `print("Olá, Mundo!")`, tem uma função bem simples: ele imprime (mostra) a mensagem "Olá, Mundo!"na tela do terminal.

É isso! Parece algo pequeno, mas é o primeiro passo para aprender a conversar com o computador.

Apesar de ser uma linha de código tão simples, ela já nos ensina um dos fundamentos mais importantes da programação: a saída de dados — ou seja, como fazer o computador apresentar informações para nós.

Mas e os programas mais complexos?

Acredite: todos os programas, por mais avançados ou sofisticados que sejam, são compostos pelas mesmas estruturas básicas. Programar é como construir com blocos de montar: aprendemos a usar essas peças fundamentais, e, aos poucos, conseguimos criar coisas incríveis!

1.5 Estruturas que compõem um programa

Aqui estão as estruturas que formam a base de qualquer programa:

Componentes de um programa

- **Variáveis**: Para armazenar informações.
- **Entrada de dados**: Para capturar dados do usuário ou de outras fontes.
- **Saída de dados**: Para exibir informações (como o nosso "Olá, Mundo!").
- **Processamento**: Realizar cálculos e transformar dados.
- **Estruturas condicionais (if)**: Para tomar decisões com base em condições.
- **Laços de repetição** (for, while, do-while): Para executar uma ação várias vezes.
- **Funções**: Blocos de código reutilizáveis que podem ser chamados várias vezes.
- **Classes**: Para organizar e estruturar programas maiores (usando conceitos de orientação a objetos).
- **Tratamento de exceções**: Para lidar com erros de forma controlada.
- **Leitura e escrita de arquivos**: Para salvar ou acessar informações armazenadas no computador.

Cuidado! não se preocupe em decorar tudo agora, com o tempo você pega o jeito.

Essas peças são como os "ingredientes básicos"da programação. Com elas, você poderá criar desde simples scripts até sistemas complexos.

1.6 Python como calculadora

Antes de introduzirmos o conceito de variáveis, vamos explorar um dos usos mais simples e práticos do Python: como uma calculadora! Isso permite que você execute operações matemáticas diretamente e veja os resultados na hora.

1.6.1 Passo 1: Abrindo o Python no Modo Interativo

No Windows, pressione a tecla Windows e digite "python"na barra de busca. Abra o programa Python 3.X (a versão que você instalou). Isso abrirá o modo interativo, onde você pode digitar comandos e ver os resultados instantaneamente.

1.6.2 Passo 2: Testando Operações Matemáticas

Digite os comandos abaixo e observe os resultados. Esses exemplos mostram diferentes operações que você pode fazer no Python:

```
3 + 3                # soma
4 * 3                # multiplicacaoo
12 / 4               # divisao (resultado com casas decimais)
12 // 4              # divisao inteira
3 ** 2               # potencia (3 elevado a 2)
26 - 5               # subtracao
3 * 4 - 2 + 13 / 3   # combinacao de operacoes: nesse caso a ordem segue o acronimo PEMDAS (mais sobre isso a seguir.)
3 * (4 - 2)          # uso de parenteses para alterar a ordem das operacoes.
```

Código 1.2: Python como calculadora.

Observação

sempre que você encontrar um # em um código python, saiba que trata-se de um comentário: um trecho de código que não vai ser executado e é completamente ignorado pelo *interpretador python* — que é, a grosso modo, o "motor"responsável por executar o código.
Pondo de forma simples, o comentário serve pra ajudar o(s) programador(es) apenas, a máquina não se importa com ele!

1.6.3 O que aprendemos com isso?

O Python segue a ordem de precedência matemática, como em uma calculadora comum. Multiplicações e divisões são realizadas antes de somas e subtrações, a menos que você use parênteses para forçar uma ordem diferente.

Você pode experimentar diferentes números e combinações de operações para entender como o Python processa cada comando.

Dica: Se você vir algo como `>>>` no terminal, é o Python esperando pelo próximo comando. Aproveite e teste à vontade!

1.7 Variáveis

O que são? Onde vivem? Para que servem? Tudo isso será respondido.

> *"Tudo será revelado"* — Mortred, Warcraft 3

O que são variáveis?

Você pode pensar em uma variável como um recipiente — como um balde, pote ou caixa — que serve para armazenar algo. Cada variável possui:

- Um **tipo**: Define o que a variável pode armazenar (números, texto, valores booleanos, etc.).
- Um **nome**: A identificação da variável, que você escolhe.
- Um **valor**: O conteúdo que ela armazena.

Por exemplo:

```
1 num = 3
```

Código 1.3: código simples

Aqui:

- O **tipo** é um inteiro (número sem casas decimais).
- O **nome** da variável é num.
- O **valor** armazenado é 3.

Isso permite realizar operações e até alterar o valor armazenado na variável. Por exemplo, poderíamos somar algo a num ou redefinir seu valor.

Onde *"vivem"* as variáveis?

As variáveis "vivem"na memória do computador, mas não no HD ou SSD! Esses são exemplos de memórias secundárias, usadas para armazenamento permanente. As variáveis de programas em execução vivem na memória RAM (Random Access Memory ou Memória de Acesso Aleatório) — Essa é a memória volátil usada enquanto o programa está rodando.

1.7.1 Nomes de variáveis: o poder está em suas mãos!

Uma das coisas legais sobre variáveis é que você pode dar o nome que quiser a elas, desde que siga algumas regras básicas do Python, basicamente: começar

com uma letra ou sublinhado, sem espaços.

Por exemplo:

```
rock_n_roll_eh_bom = True
print(type(rock_n_roll_eh_bom))
# devera mostrar <class 'bool'>
# ou seja, um tipo booleano, mais sobre classes adiante!
```

Código 1.4: exemplo de variável

Aqui, criamos uma variável com um nome bem expressivo!

- O **nome** da variável é rock_n_roll_eh_bom
- O **tipo** é booleano (True ou False).
- O **valor** é True (sim, rock'n'roll é bom!).

Isso mostra como nomes de variáveis podem tornar seu código mais legível e divertido!

De fato, você pode até verificar o tipo da variável usando a função type como demonstrado no código acima! não apenas veja o código, tente executá-lo na sua máquina também!

Resumo:

Variáveis são recipientes que armazenam valores. Elas têm um tipo, nome e valor. Vivem na memória RAM enquanto o programa está rodando. Você pode escolher nomes significativos para elas, tornando seu código mais claro e até criativo.

Observação

Embora tecnicamente seja possível usar acentos ou caracteres especiais em nomes de variáveis, não é recomendado! Isso pode causar problemas, especialmente ao trabalhar com bibliotecas externas ou ambientes que não reconhecem esses caracteres corretamente. O autor deste livreto recomenda fortemente que você aprenda inglês e não cometa heresias ao nomear suas variáveis! Este livro é apenas o começo de uma boa jornada!

1.7.2 Quais tipos existem no python?

A resposta curta é que existem seis tipos, por padrão na linguagem.

A resposta longa é: quantos você quiser.

Vamos aos seis tipos principais, são eles:

Lista de tipos principais

- **Booleano** — aceita apenas True [Verdadeiro] ou False [Falso]
- **Inteiro** — número inteiro
- **Float** — ponto flutuante (vulgarmente conhecido como decimal)
- **String** — texto, ou "cadeia de caracteres"
- **None** — um tipo que representa "vazio"ou ausência de valor
- **Complexo** — um tipo específico pra trabalhar com números complexos, não exploraremos neste livro.

para tornar toda essa discussão sobre tipos mais "palpável", tente executar os códigos abaixo.

```
# Inteiros
print(type(200))
print(type(0))
print(type(-4400))

# Booleanos
print(type(True))
print(type(False))

# Ponto flutuante
print(type(3.1415))
print(type(206.24))
print(type(-30005.08))

# None
print(type(None))

# String
print(type("testando, ola mundo"))
print(type("aaaaaaaatum"))

# Complexo
# O 'j' significa a unidade imaginaria em python!
print(type(13j)) #numeros complexos precisam ser 'posfixados' com a letra j
print(type(3 + 4j)) # outro numero complexo
```

Código 1.5: Verificando os tipos de variáveis em python

Veja que, todos esses tipos aparecem como <class 'tipo'>

Pois não seria interessante se você pudesse criar seu próprio tipo? O que é uma "classe", afinal?
Calma, vamos explorar isso em capítulos futuros! Por enquanto, focaremos na programação imperativa: um comando após o outro, instruindo a máquina passo a passo.

1.7.3 E eu com os tipos?

O que eu quero demonstrar aqui, é que uma boa parte do seu trabalho como programador será receber dados, transformá-los e apresentar ao usuário algo de relevante com eles, considere o seguinte trecho de código:

```
1 idade_do_joao = 14
2 # quantos anos joao tera daqui a 8 anos?
3 idade_do_joao = idade_do_joao + 8
4 print("joao tera", idade_do_joao, "anos, daqui a 8 anos")
```

Código 1.6: uma conta simples

Aqui, acontece que você está somando 8 a uma variável que já é um número inteiro, ou seja, não é necessário fazer nenhum tipo de conversão — dentro do print, a variável `idade_do_joao` é exibida **concatenada** (junta) as strings, de forma a compor uma string homogênia: `"Joao tera 22 anos, daqui a 8 anos"`

Suponha, no entanto, que eu queira receber do próprio usuário qual a idade dele, e, só então, calcular quantos anos ele terá daqui a 8 anos.

```
1 idade_do_usuario = int(input("Quantos anos voce tem hoje?
    "))
2 print("daqui a 8 anos, voce tera", idade_do_usuario + 8)
```

Código 1.7: Calculando a idade futura do usuário

Salve o código acima como "idade_futura_do_usuario.py"e execute ele, se tudo correr bem, o programa deve pedir a idade do usuario, você vai digitar um número qualquer, do tipo "25"(sem aspas), e ele vai mostrar quantos anos você terá daqui 8 anos, que no caso seria 33.

Nesse código, introduzimos 2 coisas diferentes: estamos fazendo uma conversão de tipos, um *type casting*, no valor que de entrada que o usuário informou através da função input e, subsequentemente atribuindo esse valor, agora convertido pra inteiro, pra variável idade_do_usuario.

Basicamente, ao executar esse código o programa vai perguntar a idade com a seguinte mensagem: "Quantos anos você tem hoje?", depois disso, ele vai converter essa entrada de dados para um inteiro, portanto, nesse primeiro momento, o usuário não pode digitar algo como "eu tenho 15", pois o programa espera somente um número — em outras palavras, quaisquer coisas digitadas naquele momento que não seja **um** número, ocasionarão em uma exceção, um erro de valor (mais sobre exceções adiante!)

Observação

Isso que fizemos no Código 1.7 é denominado **composição de funções**. Repare: uma função lê a entrada do teclado, a outra converte para inteiro, as duas juntas permitem o calculo e a apresentação para o usuário.

Isso é extremamente comum no mundo da programação, você não precisa ser nenhum gênio da matemática pra compor suas funções.

Capítulo 2

Entrada e saída de dados

2.1 Experimente coisas, quebre códigos!

Todos nós gostamos quando o código funciona, quando os nossos problemas são finalmente resolvidos e, enfim podemos descansar a cabeça. Isso é bom, mas não é tudo! Uma boa parte do seu trabalho enquanto programador vai ser garantir que o código seja seguro e corrigir os problemas, bugs ou vulnerabilidades do seu programinha, script ou sistema.

Isto posto, é melhor que você quebre logo, tantas vezes quanto forem necessárias, o seu código - é melhor falhar desde já, enquanto você pode, pois isso te ajuda a ganhar confiança com a ferramenta.

Teste: e se você esquecer de fechar um parênteses no print? e se você esquecer as aspas numa string? e se você mover parte do código de lugar, ou mesmo mandar printar uma variável que ainda não foi definida? que erro ele deu? como arruma? FALHE MUITO! Jogue os erros no Google, tente entender o porquê das coisas acontecerem da maneira que elas acontecem, adquira experiência, pois é isso que o mercado valoriza, e ninguém pode fazer por você!

Curiosidade

Existem hackers que vivem de descobrir brechas (vulnerabilidades) em sistemas e reportá-las — sim, isso dá dinheiro e é bem valorizado no mercado lá fora.
Esses profissionais agem com ética e responsabilidade, e é importante lembrar que o conhecimento em programação deve ser usado para o bem!

Um exemplo é o hacker e YouTuber Gabriel Pato — fica como sugestão dar uma olhada no conteúdo dele. O cara é monstro!

2.2 As várias formas de printar

Existem diversas formas de imprimir na tela e *interpolar* variáveis. Vejamos algumas a seguir:

```
1 print("Ola, mundo")  # sem variaveis
2 meu_numero = 10
3 print("O valor da variavel eh %i" % meu_numero)
4 print("O valor da variavel eh {}".format(meu_numero))
5 print("O valor da variavel eh", meu_numero)
6 print("O valor da variavel eh " + str(meu_numero))
7 print(f"O valor da variavel eh {meu_numero}")
```

Código 2.1: Os vários jeitos de printar em Python

Note como não é possível concatenar strings com o operador '+' sem fazer a conversão de tipos! Perceba também que, ao concatenar utilizando vírgulas, você não precisa converter os tipos — e, de quebra, o Python ainda adiciona espaços automaticamente entre os itens!

Mas qual delas eu deveria usar? — você se pergunta.

A resposta que eu vou lhe dar é: use a maneira com que você se sentir mais confortável. Eu, particularmente, sempre opto por usar uma f-string (repararam como deixei o melhor por último?). Aliás, essa é a forma mais popular de *interpolar* variáveis no Python 3.

Observação

Interpolar significa dizer para o código: "quando você for imprimir, substitua a variável que eu sinalizei pelo valor que ela carrega, beleza?"

2.2.1 Sequências de escape

Eu poderia parar por aqui a nossa discussão sobre a função `print`, e já estaria de bom tamanho — mas ela não estaria completa se não abordássemos os dois tópicos que dão nome a esta (sub)seção.

Para melhor ilustrar o que é uma sequência de escape, suponha que eu queira imprimir na tela uma tabulação ou uma nova linha — eu certamente poderia escrever "espaços"no meu código, ou pular linhas com a tecla Enter. O problema disso é que prejudica a legibilidade do código: e se eu quiser pular 3 linhas? E se eu quiser pular 15 linhas? E se eu colocar espaços a mais por acidente? Não é bom confiarmos no "olhômetro"quando se trata de sistemas precisos — especialmente no Python, que, como veremos adiante, é uma linguagem que depende fortemente de indentação e espaçamento!

É aqui que entram as sequências de escape (ou *escape sequences*, em inglês). Imagine que eu tenha um código específico para tabulação, outro para quebra de linha, e mais alguns outros para casos especiais — assim, eu economizo

espaço e melhoro a legibilidade do meu código, enquanto mantenho a precisão no espaçamento: uma tabulação sempre vai equivaler a 4 espaços, pois isso está cravado no código.

```
1 print("Veja so que bonito o meu codigo.\n")
2 print("E a quebra de linha que o acompanha!")
3 print("\tSe eu quiser imprimir uma tabulacao\n\tBasta
     acrescentar a sequencia de escape que faz isso!")
```

Código 2.2: Sequências de escape

Como podemos ver no Código 2.2, as sequências de escape que estou demonstrando são as duas que considero principais: `\t` para tabulação e `\n` para pular uma linha. Repare que o Python não se preocupa se você põe espaços antes ou depois dessas sequências — isso é contigo! Sempre que ele vir um `\t` ou `\n`, ele vai executar o comando correspondente.

Observação

Você pode escapar outros caracteres, como valores hexadecimais, ou mesmo aspas e a própria barra invertida \.
Para saber mais, visite `https://www.w3schools.com/python/gloss_python_escape_characters.asp`

2.2.2 Parâmetros opcionais

Mas nem só de *escape sequences* viverá o `print`! Existem também os *parâmetros opcionais* da função `print`. Parâmetro é o nome que se dá aos valores que são passados a uma função. Neste caso, são chamados de "opcionais"porque, se você não os especificar, não tem problema — eles não são obrigatórios!

Vejamos um exemplo:

```
1 print("Ola mundo", "olha so como eu posso", "especificar
     separadores", sep=' -- ')
2 print("Ola mundo", end='\t hahahah')
```

Código 2.3: Parâmetros opcionais

No Código 2.3, eu mostrei que podemos passar vários argumentos para a função `print`. De fato, só de printarmos várias strings separadas por vírgula, isso por si só já representa diferentes parâmetros sendo passados de uma vez só — mas a coisa não para por aí! Os parâmetros `sep` (separador) e `end` (final) também podem ser usados (até mesmo juntos!).

O resultado é o que você vê ao executar: `sep` é responsável por separar as strings. E o melhor de tudo é que eu posso escolher esse separador como quiser! O mesmo vale para o parâmetro `end`, que especifica o final da linha. Perceba: eu nem preciso pular uma linha, se eu não quiser!

2.3 Entrada de dados

Sobre isso, não há muito o que possa ser dito, honestamente: ou você pede que o usuário escreva a entrada de dados sem perguntar nada, ou você pergunta alguma coisa. A dica aqui é que, se você pedir a entrada de dados **sem** exibir um prompt para o usuário, talvez ele nem saiba o que deve digitar ali — ou pior, nem perceba que é para digitar algo!

Observação

Aqui estamos trabalhando estritamente com o terminal, mas existem outras formas de entrada de dados. Você pode utilizar bibliotecas como o `tkinter` ou o `PySimpleGUI`, que "desenham"interfaces gráficas na tela — mas não cobriremos isso neste livro.

Segue o código para que você possa testar!

```
1 seu_nome = input("Digite o seu nome: ")
2 um_numero_inteiro = int(input("Digite um numero inteiro
    qualquer: "))
3 entrada_sem_prompt = input() # repara como aqui ele vai
    ficar esperando, mas sem perguntar nada, o que torna
    tudo muito estranho e "silencioso"
4 # pra finalizar, podemos printar tudo
5 print(seu_nome, um_numero_inteiro, entrada_sem_prompt,sep
    =' |====| ')
```

Código 2.4: Entrada de dados

De fato, como eu não vou realizar nenhuma operação (conta) com o número, eu nem precisaria converter ele para inteiro, lembra disso quando discutimos a função print? teste a segunda linha sem a função `int()` ficaria assim

```
1 um_numero_inteiro = input("Digite um numero inteiro
    qualquer: ")
2 print("olha o seu numero inteiro sem conversao de tipos!"
    , um_numero_inteiro)
```

Código 2.5: Entrada de dados sem conversão de tipos

Capítulo 3

Operadores e Expressões

Este capítulo vai tratar de um tipo de dado bem específico: o tipo *Booleano*.

E o que é, afinal, o tipo `bool`? Bem, já comentamos anteriormente que um booleano é um tipo de dado que trabalha apenas com dois valores: `True` ou `False`.

Note que uma variável booleana é semelhante a um interruptor: ela só pode assumir dois estados, que representam coisas como ligado/desligado, sim/não, "vivo"ou "morto", "sinal verde"ou "sinal vermelho"— enfim, o tipo booleano trabalha com esse "dualismo eletrônico", um conceito essencial para a programação de computadores modernos!

3.1 Operadores de comparação

No próximo capítulo, você, leitor, aprenderá sobre controle de fluxo e estruturas condicionais. Mas, por ora, vamos focar nossos esforços em entender os operadores relacionais — e também os operadores lógicos.

São operadores de comparação: `==, !=, >, <, >=, <=`.

Respectivamente, eles significam: igualdade (sim, escreve-se com dois sinais de igual), diferente, maior que, menor que, maior ou igual, menor ou igual.

A notação utilizada no Python é chamada de *infixa*, o que quer dizer que o operador vem entre os dois valores — como nos exemplos a seguir.

```
3 == 3 # True (sim, 3 eh igual a 3)
3 == 20 # False (nao, 3 nao eh igual a 20)
10 != 10 # False (nao, 10 nao eh diferente de 10)
10 != 55 # True (sim, 10 eh diferente de 55)
5 > 8 # False (nao, 5 nao eh maior que 8)
30 > 15 # True (sim, 30 eh maior que 15)
20 < 15 # False (nao, 20 nao eh menor que 15)
32 < 40 # True (sim, 32 eh menor que 40)
30 <= 30 # True (sim, 30 eh "MENOR OU IGUAL" a 30)
```

```
30 <= 40 # True (sim, 30 eh "MENOR OU IGUAL" a 40)
30 <= 15 # False (nao, 30 nao eh "MENOR OU IGUAL" a 15)
30 >= 30 # True (sim, 30 eh "MAIOR OU IGUAL" a 30)
30 >= 40 # False (nao, 30 eh "MAIOR OU IGUAL" a 40)
30 >= 15 # True (sim, 30 eh "MAIOR OU IGUAL" a 15)
```

Código 3.1: Operadores de comparação

Sinta-se livre para executar todo o código acima — pode copiar e colar diretamente no interpretador do terminal (o Python 3 que já instalamos, lembra?). Você vai ver que funciona direitinho: ele compara os valores e retorna `True` ou `False` com base na expressão.

Dica

Brinque com os valores! Teste no terminal: `9 <= 15`, pressione *Enter* e veja o que ele retorna.
E se você escrever `320 >= 400`? O que será que acontece? Teste valores diferentes e internalize esse conhecimento!

3.2 Compondo expressões

E se eu te dissesse que você consegue "encadear"comparações? Pois é, isso é completamente possível com os operadores que vamos aprender agora: `and`, `or` e `not`. Na verdade, eles até possuem uma ordem de precedência — e talvez a forma mais simples de memorizar isso seja com um mnemônico que eu mesmo criei: `a-NOT-AND-OR`.

O operador `and` retorna `True` apenas se **ambas** as condições forem verdadeiras.

O operador `or` retorna `True` se **qualquer uma** das duas condições for verdadeira.

O operador `not` é unário! Ele inverte o valor da expressão: retorna `True` se a expressão for `False`, e vice-versa.

```
30 > 15 and 20 > 10 # True, pois ambas sao verdadeiras
30 > 15 and 20 < 10 # False, porque uma das operacoes deu
    False (o operador and precisa que as duas sejam True
   para retornar True)

10 < 20 or 5 != 5 # True, pois a primeira expressao e
   verdadeira, e o or so precisa que uma seja verdadeira!

not False # True
not True # False

# O melhor e que podemos utilizar essas expressoes em
   variaveis, como por exemplo
```

```
10 jogador_vivo = True
11 not jogador_vivo # False
```

Código 3.2: Operadores lógicos

Dica

Você pode compor quantas operações quiser! Experimente algo como: `30 > 15 and 10 <= 26 or (not True and False)` e veja o que acontece.

E sim — como a Álgebra Booleana é um campo da matemática — você pode (e deve!) usar parênteses para controlar a ordem de prioridade!

3.3 Tabelas verdade

As tabelas verdade a seguir exemplificam bem o que acontece com cada operador lógico quando aplicado em expressões. Não se preocupe em decorá-las num primeiro momento — tente apenas entender a lógica por trás. Com o tempo, isso se tornará natural para você, leitor.

Tabela verdade do operador `and`

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Tabela verdade do operador `or`

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Tabela verdade do operador `not`

A	not A
False	True
True	False

Capítulo 4

Controle de Fluxo

É agora que a coisa fica mais interessante! Com as estruturas condicionais, nós podemos "legislar" sobre o código — ou seja, podemos tomar decisões baseadas em condições. Por exemplo, você pode implementar um controle de acesso e decidir quem pode ou não entrar no seu sistema.

Um exemplo que sempre tenho em mente é o da Steam (uma loja virtual de jogos eletrônicos). Nesse site, há vários jogos restritos por faixa etária. Então, se você acessa um jogo proibido para menores de 14 anos, o site pede sua data de nascimento e — provavelmente — faz as contas internamente, usando o horário do sistema, para verificar se você pode visualizar o conteúdo daquele jogo (imagens, vídeos, avaliações etc.).

Certamente existem exemplos melhores, mas perceba: a condicional está ali! Não é difícil imaginar que, em algum lugar na lógica do site (ou do aplicativo), existe uma função `calcular_idade()` que faz as contas com base na sua data de nascimento e a data atual — e outra função chamada `permitir_acesso()` que retorna `True` se o usuário tiver idade suficiente, ou `False` caso contrário.

4.1 Condicionais

Sem mais delongas, vamos ao código:

```
1 permitir_acesso = True
2 if permitir_acesso == True:
3         print("Pode entrar!")
```

Código 4.1: Estrutura condicional simples

Esse código poderia ser levemente abreviado, mas optamos por mantê-lo em sua forma mais simples para fins didáticos.

De todo modo — o `if`, que em português significa "se", é exatamente isso: se o que estiver dentro da condição for `True` (verdadeiro), ele executa todas as

instruções que estiverem "dentro do if"(ou seja, que estiverem indentadas — também chamado de aninhamento).

E se a expressão dentro da condicional for `False`? Ora, então ele não executa nada! Em outras palavras: da forma como está descrito no Código 4.1, se a variável `permitir_acesso` (na linha 1) fosse `False`, tudo que está dentro do `if` seria sumariamente ignorado.

Observação

A indentação é uma parte essencial da sintaxe em Python — especialmente ao trabalhar com condicionais.
Sempre que você abrir um bloco de código com `if`, `else`, `elif`, ou outras estruturas, é obrigatório usar dois pontos (`:`) no final da linha, e em seguida indentar o bloco que será executado.
Se você tiver um `if` dentro de outro `if` (ou seja, um *if aninhado*), será necessário indentar duas vezes — uma para cada nível.
Isso vale não só para condicionais, mas também para outros blocos de código, como os laços de repetição (que veremos em breve)!

Exemplo rápido:

```
1 if condicao1:
2         print("nivel 1")
3         if condicao2:
4                 print("nivel 2")
5
```

Código 4.2: Condicionais aninhadas

4.2 Complementos

Existem dois complementos importantes quando se trata de condicionais. Apesar de não serem obrigatórios, às vezes eles se fazem necessários. São eles: `elif` e `else`.

O `elif` (uma contração de `else if`) significa "senão se", e é usado para testar uma nova condição caso a anterior não tenha sido satisfeita. Já o `else` representa um "caso contrário", ou simplesmente "senão".

Exemplificando:

```
1 idade = 15
2 if idade < 0:
3         print("Voce ainda nem existe!!")
4 elif idade < 12:
5         print("Voce eh uma crianca")
6 elif idade < 18:
7         print("Voce eh um adolescente")
8 elif idade < 60:
```

```
9           print("Voce eh um adulto")
10  else:
11          print("Voce eh um idoso")
```

Código 4.3: Condicionais aninhadas

Observação

Os blocos `elif` são opcionais, e você pode ter quantos quiser. Já o `else`, quando usado, deve sempre vir por último.
Vale lembrar: você pode escrever um `if` sozinho, ou com `if + elif`, ou `if + else` — mas nunca pode usar `else` ou `elif` sem um `if` antes.

Importante: em uma cadeia de `if/elif/else`, assim que uma das condições for satisfeita e seu bloco for executado, todas as demais instruções da cadeia são ignoradas. Ou seja, se um `elif` for executado, o Python não continuará testando os outros `elif`s nem o `else`!

Capítulo 5

Estruturas de repetição

5.1 Laço `while`

Programação e automação são áreas muito próximas: o programador, afinal, é um ser preguiçoso — e isso não é ruim! O objetivo é sempre poupar tempo e otimizar o trabalho. Graças à tecnologia, usamos retroescavadeiras em vez de pás ou (Deus me livre!) colheres para mover terra.

Veja o seguinte trecho de código:

```
print("Print de numero 1")
print("Print de numero 2")
print("Print de numero 3")
print("Print de numero 4")
print("Print de numero 5")
```

Código 5.1: Printando repetidas vezes

E é aqui que a mágica acontece! Apesar de funcionar, esse código pode ser reescrito de forma mais automática:

```
i = 1
while i <= 5:
        print(f"Print de numero {i}")
        i = i + 1
```

Código 5.2: Printando com while

No Código 5.2, declaramos uma variável chamada `i` e atribuímos a ela o valor 1. Em seguida, temos uma estrutura `while`, que funciona de forma parecida com o `if`, mas com repetição.

A palavra `while` significa "enquanto", ou seja: enquanto a condição for verdadeira, o bloco indentado será executado. Nesse caso, o Python verifica se `i <= 5` — se for verdade, ele executa o bloco: imprime a mensagem com o número atual e incrementa `i`.

Depois disso, ele volta ao início do `while` e testa novamente a condição. Isso continua até que `i` seja maior que 5, momento em que a condição se torna falsa e o laço termina.

Observação

Em algumas linguagens (ou literaturas), existe o laço `do-while`. A diferença é que o `do-while` sempre executa pelo menos uma vez, mesmo que a condição seja falsa. Já o `while` tradicional (como no Python) pode nem ser executado, dependendo da condição.
O Python não possui um laço `do-while` — trabalhamos apenas com `while` e `for`.

5.2 Laço `for`

O laço `for` é parecido com o `while`. Vamos retomar o exemplo da seção anterior para explicá-lo; posteriormente, discutiremos suas diferenças.

```
1 for i in range(1,6):
2         print(f"Print de numero {i}")
```

Código 5.3: Printando com o laço for

Assim como o `while`, o `for` também executa um bloco de código várias vezes. Dizemos que ele "itera"— ou seja, percorre um intervalo numérico ou uma lista. A diferença aqui é que usamos a função `range`, que é uma função embutida no Python, assim como o `print`.

O que estamos dizendo ao Python é: “para cada número inteiro (chamado de `i`) dentro do intervalo que começa em 1 e vai até 6, execute o bloco de código”. Importante lembrar que o valor final do `range` (no caso, o 6) **não é incluído** — os intervalos do Python são exclusivos à direita por padrão.

Lembra da *interpolação*? Pois é, ela está sendo utilizada novamente dentro do `print`. Portanto, apesar de usarem sintaxes diferentes, os laços `while` e `for` produzem o mesmo resultado nos exemplos acima.

Observação

A função `range` pode receber até 3 parâmetros. Quando recebe apenas 1, esse valor é interpretado como o "fim"do intervalo — por exemplo, `range(5)` gera os números de 0 até 4. Isso acontece porque, por padrão, o valor inicial (início do intervalo) é 0. Veremos mais sobre isso adiante.

Se você especificar 2 parâmetros, como no Código 5.3, o primeiro será o valor inicial (`start`) e o segundo será o valor final (`stop`) — lembrando que o valor final **não é incluído** no intervalo.

Por fim, é possível adicionar um terceiro parâmetro opcional, chamado `step`, que define o "passo"— ou seja, de quanto em quanto a variável será incrementada (como 1 em 1, 2 em 2, etc).

```
1 for i in range(1,30, 4):
2         print(f"Print de numero {i}")
```

Código 5.4: Outro exemplo de print com for

Capítulo 6

Trabalhando com Strings

Existem diversos métodos que podemos usar para manipular textos — mas o que é, afinal, um método? Métodos são funções associadas a classes. Lembra que vimos que existe a classe `str` (de "string")? Pois bem, agora vamos aprender como chamar os métodos que atuam sobre essa classe!

6.1 Métodos de Strings

```
1 print("ISSO AQUI VAI FICAR TUDO MINUSCULO!".lower())
2 print("e isso aqui, tudo maiusculo!!".upper())
3 print("    este metodo tira todos os espacos antes e
     depois da frase/texto !!!        ".strip())
4 print("este outro metodo converte o texto todo para uma
     LISTA".split()) # mais sobre listas no proximo
     capitulo!
5 print("todos os caracteres 'o' serao substituidos aqui
     por e".replace('o', 'e'))
6
7 minha_str = "uma pequena sequencia de texto para
     demonstrar slices"
8 print(minha_str[4:11]) # imprime a palavra "pequena",
     seguindo o intervalo do quinto caractere da string
     minha_str (comeca no 0) ate o caractere 10 (o 11 nao
     eh incluido)
9 minha_str[4] = "u" # da erro, strings no python sao
     imutaveis!
```

Código 6.1: Métodos com string

Strings também são iteráveis!

```
1 for letra in "Python":
2         print(letra)
```

Código 6.2: Iterando sobre strings

Relembrar é viver: podemos concatenar (e até repetir) strings, como mostrado abaixo:

```
1 print("Ola" + " " + "mundo!")    # Concatenacao
2 print("ha" * 3)                  # Repeticao
```

Código 6.3: Concatenação e repetição de strings

Observação

Strings em Python podem ser definidas com aspas (`"texto"`) ou com apóstrofos (`'texto'`). O importante é que o caractere de abertura e fechamento seja o mesmo.
Aliás, tanto as aspas quanto os apóstrofos podem ser "escapados"dentro da string usando a barra invertida (\). Isso permite, por exemplo, escrever aspas dentro de strings sem causar erro.
(Nota: no dia a dia, muitos chamam o apóstrofo de "aspas simples", mas tecnicamente ele é um apóstrofo mesmo!)

Capítulo 7

Listas e Coleções

Nota ao leitor

Este material está em desenvolvimento. Por enquanto, os capítulos do 7 em diante ainda estão em construção — em breve serão adicionados novos conteúdos, exemplos e exercícios.
Enquanto isso, sinta-se à vontade para explorar os capítulos já disponíveis, que cobrem os fundamentos da linguagem Python de forma gradual e prática.
Obrigado por acompanhar o projeto!

Capítulo 8

Funções

Capítulo 9

Módulos e Bibliotecas

Capítulo 10

Tratamento de Erros

Capítulo 11

Arquivos e Entrada/Saída em Disco

Capítulo 12

Introdução à Orientação a Objetos (O.O)

Licença ©