2

INTRODUÇÃO AO PYTHON

Embora cada linguagem de programação seja diferente (embora não tão diferente quanto seus projetistas querem que acreditemos), elas podem estar relacionadas em algumas dimensões.

- **Baixo nível versus alto nível** refere-se a se programamos usando instruções e objetos de dados no nível da máquina (por exemplo, mover 64 bits de dados deste local para aquele local) ou se programamos usando operações mais abstratas (por exemplo, pop abrir um menu na tela) que foram fornecidos pelo designer da linguagem.

- **Geral versus direcionado a um domínio de aplicativo** refere-se a se as operações primitivas da linguagem de programação são amplamente aplicáveis ou são ajustadas a um domínio. Por exemplo, o SQL é projetado para extrair informações de bancos de dados relacionais, mas você não gostaria de usá-lo para construir um sistema operacional.

- **Interpretado versus compilado** refere-se a se a sequência de instruções escritas pelo programador, chamada de **código-fonte,** é executada diretamente (por um interpretador) ou se é primeiro convertida (por um compilador) em uma sequência de operações primitivas no nível da máquina. (Nos primórdios dos computadores, as pessoas tinham que escrever o código-fonte em uma linguagem próxima ao **código da máquina** que pudesse ser diretamente interpretada pelo hardware do computador.) Há vantagens em ambas as abordagens. Geralmente é mais fácil depurar programas escritos em linguagens projetadas para serem interpretadas, porque o interpretador pode produzir mensagens de erro fáceis de relacionar com o código-fonte. idiomas compilados

geralmente produzem programas que rodam mais rapidamente e usam menos espaço.

Neste livro, usamos **Python.** No entanto, este livro não é sobre Python. Certamente ajudará você a aprender Python, e isso é bom. O que é muito mais importante, no entanto, é que você aprenderá algo sobre como escrever programas que resolvem problemas. Você pode transferir essa habilidade para qualquer linguagem de programação.

Python é uma linguagem de programação de propósito geral que você pode usar efetivamente para construir quase qualquer tipo de programa que não precise de acesso direto ao hardware do computador. Python não é ideal para programas que possuem altas restrições de confiabilidade (devido à sua fraca verificação semântica estática) ou que são construídos e mantidos por muitas pessoas ou por um longo período de tempo (novamente devido à fraca verificação semântica estática).

O Python tem várias vantagens sobre muitas outras linguagens.
É uma linguagem relativamente simples e fácil de aprender. Como o Python foi projetado para ser interpretado, ele pode fornecer o tipo de feedback em tempo de execução que é especialmente útil para programadores novatos. Um número grande e crescente de interfaces de bibliotecas disponíveis gratuitamente para Python e fornece funcionalidade estendida útil. Usamos várias dessas bibliotecas neste livro.

Estamos prontos para apresentar alguns dos elementos básicos do Python.
Estes são comuns a quase todas as linguagens de programação em conceito, embora não em detalhes.

Este livro não é apenas uma introdução ao Python. Ele usa o Python como um veículo para apresentar conceitos relacionados à resolução de problemas computacionais e pensamento. A linguagem é apresentada em dribs e drabs, conforme necessário para este propósito ulterior. Os recursos do Python que não precisamos para esse propósito não são apresentados. Sentimo-nos confortáveis em não cobrir todos os detalhes porque excelentes recursos on-line descrevem todos os aspectos do idioma. Sugerimos que você use esses recursos on-line gratuitos conforme necessário.

Python é uma linguagem viva. Desde a sua introdução por Guido von Rossum em 1990, passou por muitas mudanças. Durante a primeira década de sua vida, Python foi uma linguagem pouco conhecida e pouco usada.
Isso mudou com a chegada do Python 2.0 em 2000. Além de incorporar melhorias importantes à própria linguagem, marcou uma mudança no caminho evolutivo da linguagem. muitos grupos

começou a desenvolver bibliotecas que faziam uma interface perfeita com o Python, e o suporte e desenvolvimento contínuos do ecossistema Python tornaram-se uma atividade baseada na comunidade.

O Python 3.0 foi lançado no final de 2008. Esta versão do Python eliminou muitas das inconsistências no design do Python 2. No entanto, o Python 3 não é compatível com versões anteriores. Isso significa que a maioria dos programas e bibliotecas escritas para versões anteriores do Python não podem ser executadas usando implementações do Python 3.

Até agora, todas as importantes bibliotecas Python de domínio público têm foi portado para o Python 3. Hoje, não há razão para usar o Python 2.

2.1 Instalando Python e Python IDEs

Antigamente, os programadores usavam editores de texto de uso geral para inserir seus programas. Hoje, a maioria dos programadores prefere usar um editor de texto que faça parte de um **ambiente de desenvolvimento integrado (IDE).**

O primeiro Python IDE, IDLE,9 veio como parte do pacote de instalação padrão do Python. À medida que a popularidade do Python cresceu, outros IDEs surgiram. Esses IDEs mais novos geralmente incorporam algumas das bibliotecas Python mais populares e fornecem recursos não fornecidos pelo IDLE. **Anaconda** e Canopy estão entre os mais populares desses IDEs. O código que aparece neste livro foi criado e testado usando o Anaconda.

IDEs são aplicativos, como qualquer outro aplicativo em seu computador. Inicie um da mesma forma que iniciaria qualquer outro aplicativo, por exemplo, clicando duas vezes em um ícone. Todos os IDEs do Python fornecem

- Um editor de texto com realce de sintaxe, preenchimento automático e recuo inteligente,
- um shell com realce de sintaxe e um
- depurador integrado, que você pode ignorar com segurança por enquanto.

Este seria um bom momento para instalar o Anaconda (ou algum outro IDE) em seu computador, para que você possa executar os exemplos no

livro e, mais importante, tente os exercícios de programação com os dedos. Para instalar o Anaconda, vá para

https://www.anaconda.com/distribution/

e siga as instruções.

Assim que a instalação estiver concluída, inicie o aplicativo Anaconda Navigator. Uma janela contendo uma coleção de ferramentas Python aparecerá. A janela se parecerá com a Figura 2-1. 10 Por enquanto, a única ferramenta que usaremos é **o Spyder.** Quando você iniciar o Spyder (clicando no botão Launch , entre todas as coisas), uma janela semelhante à Figura 2-2 será aberta.

Figura 2-1 Janela de inicialização do Anaconda

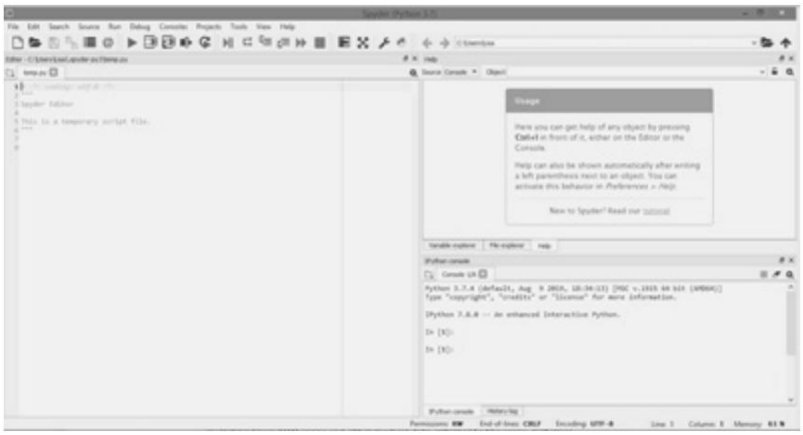

Figura 2-2 Janela do Spyder

O painel no canto inferior direito da Figura 2-2 é um **console IPython** executando um shell Python interativo . Você pode digitar e executar comandos Python nesta janela. O painel no canto superior direito é uma janela de ajuda. Muitas vezes é conveniente fechar essa janela (clicando no x), para que mais espaço esteja disponível para o console do IPython. O painel à esquerda é uma janela de edição na qual você pode digitar programas que podem ser salvos e executados. A barra de ferramentas na parte superior da janela facilita a execução de várias tarefas, como abrir arquivos e imprimir programas.11 A documentação do Spyder pode ser encontrada em https://www.spyder-ide.org/.

2.2 Os Elementos Básicos do Python

Um **programa** Python , às vezes chamado de **script,** é uma sequência de definições e comandos. O interpretador Python no shell avalia as definições e executa os comandos.

Recomendamos que você inicie um shell Python (por exemplo, iniciando o Spyder) agora e use-o para experimentar os exemplos contidos no restante deste capítulo. E, aliás, no resto do livro.

Um **comando,** muitas vezes chamado de **instrução,** instrui o interpretador a fazer algo. Por exemplo, a instrução print('Yankees rule!') instrui o interpretador a chamar a function12 print, que retorna a string Yankees rule! para a janela associada ao shell.

A sequência de comandos

```
print('Regra dos Yankees!') print('Mas não
em Boston!') print('Regra dos Yankees,', 'mas não
em Boston!')
```

faz com que o interpretador produza a saída

```
Regra dos ianques!
Mas não em Boston!
Os Yankees governam, mas não em Boston!
```

Observe que dois valores foram passados para impressão na terceira instrução. A função print pega um número variável de argumentos separados por vírgulas e os imprime, separados por um caractere de espaço, na ordem em que aparecem.

2.2.1 Objetos, expressões e tipos numéricos Objetos são as coisas principais que os programas Python manipulam. Cada objeto tem um **tipo** que define o que os programas podem fazer com aquele objeto.

Os tipos são escalares ou não escalares. Objetos **escalares** são indivisíveis. Pense neles como os átomos da linguagem.13 Objetos **não escalares ,** por exemplo, strings, têm estrutura interna.

Muitos tipos de objetos podem ser indicados por **literais** no texto de um programa. Por exemplo, o texto 2 é um literal representando um número e o texto 'abc' é um literal representando uma string.

Python tem quatro tipos de objetos escalares:

- **int** é usado para representar números inteiros. Literais do tipo int são escritos da maneira que normalmente denotamos inteiros (por exemplo,
- -3 ou 5 ou 10002). **float** é usado para representar números reais. Os literais do tipo float sempre incluem um ponto decimal (por exemplo, 3,0 ou 3,17 ou -28,72). (Também é possível escrever literais do tipo float usando notação científica. Por exemplo, o literal 1.6E3 representa 1.6*103 , ou seja, é o mesmo que 1600.0.) Você pode se perguntar por que esse tipo não é chamado de real. Dentro do computador, os valores do tipo float são armazenados como **números de ponto flutuante.** Essa representação, que é usada por todas as linguagens de programação modernas, tem muitas vantagens. No entanto, em algumas situações, faz com que a aritmética de ponto flutuante se comporte de maneiras ligeiramente diferentes da aritmética em números reais. Discutimos isso na Seção 3.3. **bool** é
- usado para representar os valores booleanos True e False.
- **None** é um tipo com um único valor. Falaremos mais sobre Nenhum na Seção 4.

Objetos e **operadores** podem ser combinados para formar **expressões,** cada uma das quais avaliada como um objeto de algum tipo. Isso é chamado de **valor** da expressão. Por exemplo, a expressão 3 + 2 denota o objeto 5 do tipo int e a expressão 3.0 + 2.0 denota o objeto 5.0 do tipo float.

O operador == é usado para testar se duas expressões são avaliadas como o mesmo valor, e o operador != é usado para testar se duas expressões são avaliadas como valores diferentes. Um único = significa algo

bastante diferentes, como veremos na Seção 2.2.2. Esteja avisado - você cometerá o erro de digitar “=” quando pretendia digitar “==”.
Fique atento a esse erro.

Em um console do Spyder, algo parecido com In [1]: é um **prompt de shell** indicando que o interpretador espera que o usuário digite algum código Python no shell. A linha abaixo do prompt é produzida quando o interpretador avalia o código Python inserido no prompt, conforme ilustrado pela seguinte interação com o interpretador:

```
3
Fora[1]: 3

3+2
Fora [2]: 5

3,0+2,0
Fora[3]: 5.0

3!=2
Out[4]: Verdadeiro
```

O tipo de função interna do Python pode ser usado para descobrir o tipo de um objeto:

```
tipo(3)
Fora[5]: int

tipo(3.0)
Fora[6]: flutuar
```

Os operadores em objetos do tipo int e float são listados na Figura 2-3. Os operadores aritméticos têm a precedência usual. Por exemplo, * vincula mais estreitamente do que +, portanto, a expressão x+y*2 é avaliada primeiro multiplicando y por 2 e, em seguida, adicionando o resultado a x. A ordem de avaliação pode ser alterada usando parênteses para agrupar subexpressões, por exemplo, (x+y)*2 primeiro adiciona x e depois multiplica o resultado por 2.

i+j is the sum of i and j. If i and j are both of type int, the result is an int. If either is a float, the result is a float.
i-j is i minus j. If i and j are both of type int, the result is an int. If either is a float, the result is a float.
i*j is the product of i and j. If i and j are both of type int, the result is an int. If either is a float, the result is a float.
i//j is floor division. For example, the value of 6//2 is the int 3 and the value of 6//4 is the int 1. The value is 1 because floor division returns the quotient and ignores the remainder. If j == 0, an error occurs.
i/j is i divided by j. The / operator performs floating-point division. For example, the value of 6/4 is 1.5. If j == 0, an error occurs.
i%j is the remainder when the int i is divided by the int j. It is typically pronounced "i mod j," which is short for "i modulo j."
ij** is i raised to the power j. If i and j are both of type int, the result is an int. If either is a float, the result is a float.

Figura 2-3 Operadores nos tipos int e float

Os operadores primitivos no tipo bool são and, or e not:

- **a e b** é verdadeiro se a e b são verdadeiros e falso caso contrário. **a ou**
- **b** é verdadeiro se pelo menos um de a ou b for verdadeiro e falso caso contrário.
- **not a** é True se a for False, e False se a for True.

2.2.2 Variáveis e variáveis de atribuição

fornecem uma maneira de associar nomes a objetos. Considere o código

```
pi = 3
raio = 11 área =
pi * (raio**2) raio = 14
```

O código primeiro **vincula** os nomes pi e radius a diferentes objetos do tipo int. 14
Em seguida, vincula a área de nome a um terceiro objeto do tipo int.
Isso é representado no lado esquerdo da Figura 2-4.

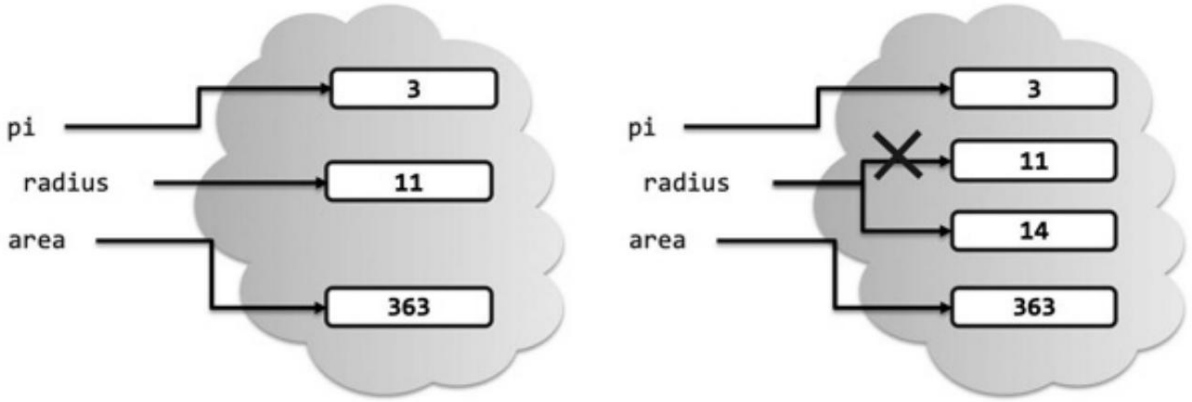

Figura 2-4 Vinculação de variáveis a objetos

Se o programa executar radius = 14, o nome radius é religado para um objeto diferente do tipo int, conforme mostrado no lado direito da Figura 2-4. Observe que esta atribuição não tem efeito sobre o valor ao qual a área está vinculada. Ele ainda está vinculado ao objeto denotado pela expressão 3*(11**2).

Em Python, ***uma variável é apenas um nome,*** nada mais. Lembre-se disso - é importante. Uma instrução **de atribuição** associa o nome à esquerda do símbolo = com o objeto denotado pela expressão à direita do símbolo = . Lembre-se disso também: um objeto pode ter um, mais de um ou nenhum nome associado a ele.

Talvez não devêssemos ter dito “uma variável é ***apenas*** um nome”. Apesar do que Juliet disse,15 nomes importam. As linguagens de programação nos permitem descrever cálculos para que os computadores possam executá-los. Isso não significa que apenas computadores leem programas.

Como você logo descobrirá, nem sempre é fácil escrever programas que funcionem corretamente. Programadores experientes confirmarão que gastam muito tempo lendo programas na tentativa de entender por que eles se comportam dessa maneira. Portanto, é de importância crítica escrever programas de modo que sejam fáceis de ler. A escolha adequada de nomes de variáveis desempenha um papel importante no aprimoramento da legibilidade.

Considere os dois fragmentos de código

```
a = 3,14159                pi = 3,14159 diâmetro
b = 11,2                   = 11,2
c = a*(b**2) área = pi*(diâmetro**2)
```

No que diz respeito ao Python, os fragmentos de código não são diferentes. Quando executados, eles farão a mesma coisa. Para um leitor humano, no entanto, eles são bem diferentes. Quando lemos o fragmento à esquerda, não há razão ***a priori*** para suspeitar que algo esteja errado.
No entanto, uma rápida olhada no código à direita deve nos levar a suspeitar de que algo está errado. A variável deveria ter sido nomeada como raio em vez de diâmetro, ou o diâmetro deveria ter sido dividido por 2,0 no cálculo da área.

Em Python, os nomes das variáveis podem conter letras maiúsculas e minúsculas, dígitos (embora não possam começar com um dígito) e o caractere especial _ (sublinhado). Os nomes das variáveis do Python diferenciam maiúsculas de minúsculas, por exemplo, Romeo e romeo são nomes diferentes. Por fim, algumas **palavras reservadas** (às vezes chamadas de palavras- **chave)** em Python que possuem significados integrados e não podem ser usadas como nomes de variáveis. Versões diferentes do Python têm listas ligeiramente diferentes de palavras reservadas. As palavras reservadas no Python 3.8 são

e	quebrar	elif	para	em	não	Verdadeiro
como	aula	outro	De é		ou	tentar
afirmar continuar exceto passagem lambda global						enquanto
definição assíncrona		Falso	se	aumento não local com		
espere pelo		finalmente importar nenhum			rendimento de retorno	

Outra boa maneira de melhorar a legibilidade do código é adicionar **comentários.** O texto após o símbolo # não é interpretado pelo Python. Por exemplo, podemos escrever

```
lado = 1 #comprimento dos lados de um quadrado unitário raio = 1 #raio
de um círculo unitário #subtrair a área do círculo unitário da
área do quadrado unitário area_circle = pi*raio**2 area_square = lado*diferença de lados =
area_square – area_circle
```

Python permite atribuição múltipla. A declaração

```
x, y = 2, 3
```

associa x a 2 e y a 3. Todas as expressões no lado direito da atribuição são avaliadas antes de quaisquer associações serem alteradas.

Isso é conveniente, pois permite que você use atribuição múltipla para trocar as ligações de duas variáveis.

Por exemplo, o código

```
x, y = 2, 3
x, y = y, x print('x
=', x) print('y =', y)
```

vai imprimir

```
x = 3
y = 2
```

2.3 Programas de ramificação

Os tipos de cálculos que examinamos até agora são chamados de **programas em linha reta.** Eles executam uma instrução após a outra na ordem em que aparecem e param quando ficam sem instruções. Os tipos de cálculos que podemos descrever com programas de linha reta não são muito interessantes. Na verdade, eles são absolutamente chatos.

Programas **de ramificação** são mais interessantes. A instrução de ramificação mais simples é uma **condicional.** Conforme mostrado na caixa da Figura 2-5, uma declaração condicional tem três partes:

- Um teste, ou seja, uma expressão avaliada como Verdadeiro ou Falso
- Um bloco de código que é executado se o teste for avaliado como True
- Um bloco de código opcional que é executado se o teste for avaliado como Falso

Depois que uma instrução condicional é executada, a execução é retomada em o código que segue a instrução.

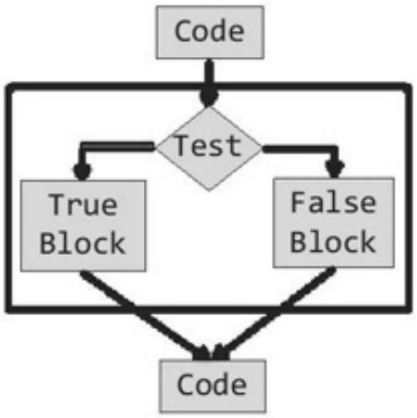

Figura 2-5 Fluxograma para declaração condicional

Em Python, uma instrução condicional tem a forma

if *Expressão booleana: bloco de código* if *Expressão booleana: bloco de código*

ou

outro:

bloco de código

Ao descrever a forma das instruções Python, usamos itálico para identificar os tipos de código que podem ocorrer naquele ponto de um programa. Por exemplo, ***a expressão booleana*** indica que qualquer expressão avaliada como True ou False pode seguir a palavra reservada if, e ***o bloco de código*** indica que qualquer sequência de instruções Python pode seguir else:.

Considere o seguinte programa que imprime “Par” se o valor da variável x for par e “Ímpar” caso contrário:

```
se x%2 == 0:
        print('Par') senão:

        print('Ímpar')
print('Feito com condicional')
```

A expressão x%2 == 0 é avaliada como True quando o resto de x dividido por 2 é 0 e False caso contrário. Lembre-se de que == é usado para comparação, pois = é reservado para atribuição.

A indentação é semanticamente significativa em Python. Por exemplo, se a última instrução no código acima for recuada,

seria parte do bloco de código associado ao else, em vez do bloco de código que segue a instrução condicional.

Python é incomum em usar indentação dessa maneira. A maioria das outras linguagens de programação usa símbolos de colchetes para delinear blocos de código, por exemplo, C inclui blocos entre colchetes, { }. Uma vantagem da abordagem Python é que ela garante que a estrutura visual de um programa seja uma representação precisa de sua estrutura semântica. Como a indentação é semanticamente importante, a noção de linha também é importante. Uma linha de código muito longa para ser lida facilmente pode ser dividida em várias linhas na tela terminando cada linha na tela, exceto a última, com uma barra invertida (\). Por exemplo,

```
  x = 11111111111111111111111111111111 +
22222222222233322222222 +\
        333333333333333333333333333333
```

Linhas longas também podem ser agrupadas usando a continuação de linha implícita do Python. Isso é feito com colchetes, ou seja, parênteses, colchetes e colchetes. Por exemplo,

```
  x = 11111111111111111111111111111111 +
22222222222233322222222 +
        333333333333333333333333333333
```

é interpretado como duas linhas (e, portanto, produz um erro de sintaxe de "recuo inesperado", enquanto

```
  x = (11111111111111111111111111111111 +
22222222222233322222222 +
        333333333333333333333333333333)
```

é interpretado como uma única linha por causa dos parênteses. Muitos programadores Python preferem usar continuações de linha implícitas a usar uma barra invertida. Mais comumente, os programadores quebram linhas longas em vírgulas ou operadores.

Voltando aos condicionais, quando o bloco verdadeiro ou o bloco falso de um condicional contém outro condicional, diz-se que as instruções condicionais estão **aninhadas.** O código a seguir contém condicionais aninhadas em ambas as ramificações da instrução if de nível superior .

```
se x%2 == 0:
        se x%3 == 0:
                print('Divisivel por 2 e 3')
```

```
        outro:
                print('Divisível por 2 e não por 3')
elif x%3 == 0:
        print('Divisível por 3 e não por 2')
```

O elif no código acima significa “else if”.

Muitas vezes é conveniente usar uma **expressão booleana composta** no teste de uma condicional, por exemplo,

```
if x < y e x < z: print('x é o
        menor') elif y < z: print('y é o
menor') else:


        print('z é o menor')
```

Exercício de dedo: Escreva um programa que examine três variáveis — e z — e x, e, imprima o maior número ímpar entre elas. Se nenhum deles são ímpares, deve imprimir o menor valor dos três.

Você pode atacar este exercício de várias maneiras. Há oito casos separados a serem considerados: todos são ímpares (um caso), exatamente dois deles são ímpares (três casos), exatamente um deles é ímpar (três casos) ou nenhum deles é ímpar (um caso). Portanto, uma solução simples envolveria uma sequência de oito instruções if , cada uma com uma única instrução print :

```
se x%2 != 0 e y%2 != 0 e z%2 != 0: print(max(x, y, z)) se
        x%2 != 0 e y%2 != 0 e z% 2
== 0: imprimir(max(x, y)) se x%2 != 0 e y%2 == 0 e z%2 !
        = 0: imprimir(max(x, z))
se x%2 = = 0 e y%2 != 0 e z%2 != 0: print(max(y, z)) se
        x%2 != 0 e y%2 == 0 e
z%2 == 0: print( x) se x%2 == 0 e y%2 != 0 e z%2 == 0:
        print(y) se x%2 == 0 e
y%2 == 0 e z%2 != 0: print(z) se x%2 == 0 e y%2 == 0 e
        z%2 == 0:
print(min(x, y, z))
```

Isso faz o trabalho, mas é bastante complicado. Não são apenas 16 linhas de código, mas as variáveis são repetidamente testadas quanto à estranheza. O código a seguir é mais elegante e mais eficiente:

```
resposta = min(x, y, z) se x%2 !=
0:
        resposta = x
se y%2 != 0 e y > resposta: resposta = y se
        z%2 != 0 ez >
resposta:
        resposta = z
imprimir(resposta)
```

O código é baseado em um paradigma de programação comum. Ele começa atribuindo um valor provisório a uma variável (resposta), atualizando-o quando apropriado e, em seguida, imprimindo o valor final da variável. Observe que ele testa se cada variável é ímpar exatamente uma vez e contém apenas uma única instrução de impressão. Esse código é o melhor que podemos fazer, pois qualquer programa correto deve verificar cada variável quanto à estranheza e comparar os valores das variáveis ímpares para encontrar o maior deles.

Python suporta **expressões condicionais ,** bem como declarações condicionais. Expressões condicionais são da forma

expr1 se *condição* senão *expr2*

Se a condição for avaliada como True, o valor da expressão inteira é *expr1;* caso contrário, é *expr2.* Por exemplo, a declaração

```
x = y se y > z senão z
```

define x ao máximo de e z. e Uma expressão condicional pode aparecer em qualquer lugar em que uma expressão comum possa aparecer, inclusive dentro de expressões condicionais. Assim, por exemplo,

```
print((x se x > z senão z) se x > y senão (y se y > z senão z))
```

imprime o máximo de x, e, e z.

Os condicionais nos permitem escrever programas que são mais interessantes do que os programas de linha reta, mas a classe de programas de ramificação ainda é bastante limitada. Uma maneira de pensar sobre o poder de uma classe de programas é em termos de quanto tempo eles podem levar para serem executados. Suponha que cada linha de código leve uma unidade de tempo para ser executada. Se uma linha reta

programa tiver n linhas de código, levará n unidades de tempo para ser executado. E quanto a um programa de ramificação com n linhas de código? Pode levar menos de n unidades de tempo para ser executado, mas não pode demorar mais, pois cada linha de código é executada no máximo uma vez.

Diz-se que um programa para o qual o tempo máximo de execução é limitado pela duração do programa é executado em **tempo constante.** Isso não significa que cada vez que o programa é executado, ele executa o mesmo número de etapas. Isso significa que existe uma constante, k, de modo que o programa não levará mais do que k passos para ser executado. Isso implica que o tempo de execução não cresce com o tamanho da entrada do programa.

Os programas de tempo constante são limitados no que podem fazer. Considere escrever um programa para contar os votos em uma eleição. Seria realmente surpreendente se alguém pudesse escrever um programa que pudesse fazer isso em um tempo que fosse independente do número de votos expressos. Na verdade, é impossível fazê-lo. O estudo da dificuldade intrínseca dos problemas é o tema da **complexidade computacional.** Voltaremos a esse tópico várias vezes neste livro.

Felizmente, precisamos apenas de mais uma construção de linguagem de programação, a iteração, para nos permitir escrever programas de complexidade arbitrária. Chegamos a isso na Seção 2.5.

2.4 Strings e Entrada

Objetos do tipo str são usados para representar caracteres.16 Literais do tipo str podem ser escritos usando aspas simples ou duplas, por exemplo, 'abc' ou "abc". O literal '123' denota uma string de três caracteres, não o número 123.

Tente digitar as seguintes expressões no interpretador Python.

```
'a'
3*4
3*'a'
3+4
'um'+'um'
```

Diz-se que o operador + está **sobrecarregado** porque tem significados diferentes, dependendo dos tipos de objetos aos quais é aplicado. Por exemplo, o operador + significa adição quando aplicado

a dois números e concatenação quando aplicada a duas strings. O operador * também está sobrecarregado. Significa o que você espera que signifique quando seus operandos são ambos números. Quando aplicado a um int e um str, é um **operador de repetição — a** expressão n*s, onde n é um int e s é um str, resulta em um str com n repetições de s. Por exemplo, a expressão 2*'John' tem o valor 'JohnJohn'. Existe uma lógica nisso. Assim como a expressão matemática 3*2 equivale a 2+2+2, a expressão 3*'a' equivale a 'a'+'a'+'a'.

Agora tente digitar

```
new_id
'a'*'a'
```

Cada uma dessas linhas gera uma mensagem de erro. A primeira linha produz a mensagem

```
NameError: o nome 'new_id' não está definido
```

Como new_id não é um literal de nenhum tipo, o interpretador o trata como um nome. No entanto, como esse nome não está vinculado a nenhum objeto, tentar usá-lo causa um erro de tempo de execução. O código 'a'*'a' produz a mensagem de erro

```
TypeError: não é possível multiplicar a sequência por não int do tipo 'str'
```

Essa **verificação de tipo** existe é uma coisa boa. Ele transforma erros descuidados (e às vezes sutis) em erros que interrompem a execução, em vez de erros que levam os programas a se comportarem de maneiras misteriosas. A verificação de tipo em Python não é tão forte quanto em algumas outras linguagens de programação (por exemplo, Java), mas é melhor em Python 3 do que em Python 2. Por exemplo, está claro o que < deve significar quando é usado para comparar dois strings ou dois números. Mas qual deve ser o valor de '4' < 3 ? De forma bastante arbitrária, os projetistas do Python 2 decidiram que deveria ser False, porque todos os valores numéricos deveriam ser menores que todos os valores do tipo str. Os projetistas do Python 3 e da maioria das outras linguagens modernas decidiram que, como essas expressões não têm um significado óbvio, elas deveriam gerar uma mensagem de erro.

Strings são um dos vários tipos de sequência em Python. Eles compartilham as seguintes operações com todos os tipos de sequência.

- O **comprimento** de uma string pode ser encontrado usando a função len . Por exemplo, o valor de len('abc') é 3.
- **A indexação** pode ser usada para extrair caracteres individuais de uma string. Em Python, toda indexação é baseada em zero. Por exemplo, digitar 'abc'[0] no interpretador fará com que ele exiba a string 'a'. Digitar 'abc'[3] produzirá a mensagem de erro IndexError: Desde que o índice de string fora do intervalo. Python usa 0 para indicar o primeiro elemento de uma string, o último elemento de uma string de comprimento 3 é acessado usando o índice 2. Números negativos são usados para indexar a partir do final de uma string. Por exemplo, o valor de 'abc'[ÿ1] é 'c'.
- **O fatiamento** é usado para extrair substrings de comprimento arbitrário. Se s for uma string, a expressão s[start:end] denota a substring de s que começa no início do índice e termina no final do índice-1. Por exemplo, 'abc'[1:3] resulta em 'bc'. Por que termina no índice end-1 em vez de terminar? Assim, expressões como 'abc'[0:len('abc')] têm o valor esperado. Se o valor antes dos dois pontos for omitido, o padrão será 0. Se o valor após os dois pontos for omitido, o padrão será o comprimento da string. Conseqüentemente, a expressão 'abc'[:] é semanticamente equivalente à mais detalhada 'abc'[0:len('abc')]. Também é possível fornecer um terceiro argumento para selecionar uma fatia não contígua de uma string. Por exemplo, o valor da expressão '123456789'[0:8:2] é a string '1357'.

Geralmente é conveniente converter objetos de outros tipos em strings usando a função str . Considere, por exemplo, o código

```
num = 30000000 fração
= 1/2 print(num*fração,
'é', fração*100, '%', 'de', num) print(num*fração, 'é', str(fração*100) + ' %', 'de', num)
```

que imprime

```
15000000,0 é 50,0% de 30000000 15000000,0 é 50,0%
de 30000000
```

A primeira instrução print insere um espaço entre 50 e % porque o Python insere automaticamente um espaço entre os argumentos a serem impressos. A segunda instrução print produz uma saída mais apropriada combinando 50 e % em um único argumento do tipo
estr.

As conversões de tipo (também chamadas de **conversões de tipo)** são usadas com frequência no código Python. Usamos o nome de um tipo para converter valores para esse tipo. Então, por exemplo, o valor de int('3')*4 é 12. Quando um float é convertido em um int, o número é truncado (não arredondado), por exemplo, o valor de int(3.9) é o int 3 .

Voltando à saída de nossas declarações de impressão, você pode estar se perguntando sobre aquele .0 no final do primeiro número impresso. Isso aparece porque 1/2 é um número de ponto flutuante e o produto de um int e um float é um float. Isso pode ser evitado convertendo num*fraction em um int. O código

```
print(int(num*fração), 'é', str(fração*100) + '%', 'de', num)
```

imprime 15000000 é 50,0% de 30000000.

O Python 3.6 introduziu uma maneira alternativa e mais compacta de criar expressões de string. Uma **string f** consiste no caractere f (ou F) seguido por um tipo especial de literal de string chamado **literal de string formatado.** Os literais de cadeia de caracteres formatados contêm sequências de caracteres (como outros literais de cadeia de caracteres) e expressões entre colchetes. Essas expressões são avaliadas em tempo de execução e convertidas automaticamente em strings. O código

```
print(f'{int(num*fração)} é {fração*100}% de {num}')
```

produz a mesma saída que a instrução print anterior, mais detalhada. Se você quiser incluir uma chave na string denotada por uma string f, use duas chaves. Por exemplo, print(f'{{{3*5}}}') imprime {15}.

A expressão dentro de uma string f pode conter modificadores que controlam a aparência da string de saída.17 Esses modificadores são separados da expressão que indica o valor a ser modificado por dois pontos. Por exemplo, a string f f'{3.14159:.2f}' é avaliada como a string '3.14' porque o modificador .2f instrui o Python a truncar a representação de string de um número de ponto flutuante para dois dígitos após o ponto decimal. E a declaração

```
print(f'{num*fração:,.0f} é {fração*100}% de {num:,}')
```

imprime 15.000.000 é 50,0% de 30.000.000 porque o modificador instrui,o Python a usar vírgulas como separadores de milhar. Apresentaremos outros modificadores convenientes posteriormente neste livro.

2.4.1 Entrada

Python 3 tem uma função, **entrada,** que pode ser usada para obter entrada diretamente de um usuário. A função de entrada recebe uma string como argumento e a exibe como um prompt no shell. A função então espera que o usuário digite algo e pressione a tecla Enter . A linha digitada pelo usuário é tratada como uma string e se torna o valor retornado pela função.

Executando o código name = input('Enter your name: ') exibirá a linha

```
Digite seu nome:
```

na janela do console. Se você digitar George Washington e pressionar enter, a string 'George Washington' será atribuída à variável Se você executar print('Are you really', nome. name, '?'), o
linha

```
Você é realmente George Washington?
```

seria exibido. Observe que a instrução print introduz um espaço antes do “?”. Ele faz isso porque quando print recebe vários argumentos, ele coloca um espaço entre os valores associados aos argumentos. O espaço pode ser evitado executando + name + '?') ou print(f'Are you really
print('Você é mesmo '
{name}?'), cada um dos quais produz uma única string e passa essa string como o único argumento a ser impresso.

Agora considere o código

```
n = input('Digite um int: ') print(type(n))
```

Este código sempre será impresso

```
<class 'str'>
```

porque input sempre retorna um objeto do tipo str, mesmo que o usuário tenha digitado algo que se pareça com um número inteiro. Por exemplo, se o usuário tivesse inserido 3, n seria vinculado ao str '3' e não ao int 3. Portanto, o valor da expressão n*4 seria '3333' em vez de 12. A boa notícia é que sempre que uma string é um literal válido de algum tipo, uma conversão de tipo pode ser aplicada a ela.

Exercício de dedo: Escreva um código que peça ao usuário para inserir seu aniversário no formato mm/dd/aaaa e, em seguida, imprima uma string no formato 'Você nasceu no ano aaaa'.

2.4.2 Uma digressão sobre codificação de caracteres Por muitos anos, a maioria das linguagens de programação usou um padrão chamado ASCII para a representação interna de caracteres. Esse padrão incluía 128 caracteres, o suficiente para representar o conjunto usual de caracteres que aparecem no texto em inglês — mas não o suficiente para abranger os caracteres e acentos que aparecem em todos os idiomas do mundo.

O padrão **Unicode** é um sistema de codificação de caracteres projetado para suportar o processamento digital e exibição de textos escritos de todos os idiomas. O padrão contém mais de 120.000 caracteres, abrangendo 129 scripts modernos e históricos e vários conjuntos de símbolos.
O padrão Unicode pode ser implementado usando diferentes codificações internas de caracteres. Você pode dizer ao Python qual codificação usar inserindo um comentário do formulário

```
# -*- codificação: nome da codificação -*-
```

como a primeira ou segunda linha do seu programa. Por exemplo,

```
# -*- codificação: utf-8 -*-
```

instrui o Python a usar UTF-8, a codificação de caracteres usada com mais frequência para páginas da ~~We~~b.18 Se você não tiver esse comentário em seu programa, a maioria das implementações do Python será padronizada para UTF-8.

Ao usar UTF-8, você pode, se o editor de texto permitir, inserir diretamente o código como

```
print('Mluvíš anglicky?') print('Do you
speak English?')
```

que vai imprimir

```
Você fala inglês?
    você fala inglês
```

Você deve estar se perguntando como consegui digitar a string 'ÿ ÿÿ ÿÿÿÿÿÿ ÿÿÿÿÿÿ?'. Eu não. Como a maior parte da web usa UTF-8, consegui cortar a string de uma página da web e colá-la diretamente em meu programa. Existem maneiras de inserir caracteres Unicode diretamente de um teclado, mas, a menos que você tenha um teclado especial, todas elas são bastante complicadas.

2.5 Loops while

Perto do final da Seção 2.3, mencionamos que a maioria das tarefas computacionais não pode ser realizada usando programas de ramificação. Considere, por exemplo, escrever um programa que pede o número de X's. Você pode pensar em escrever algo como

```
num_x = int(input('Quantas vezes devo imprimir a letra X? ')) to_print = if num_x
== 1:                 "
to_print = 'X'
elif num_x == 2:
        to_print = 'XX' elif
num_x == 3: to_print
        = 'XXX'



#… print(to_print)
```

Mas rapidamente se tornaria aparente que você precisaria de tantos condicionais quantos inteiros positivos - e há um número infinito deles. O que você deseja escrever é um programa parecido com (o seguinte é **pseudocódigo,** não Python)

```
num_x = int(input('Quantas vezes devo imprimir a letra X? ')) to_print = #
                      "
concatenar
X para to_print num_x vezes print(to_print)
```

Quando queremos que um programa faça a mesma coisa várias vezes, podemos usar **a iteração.** Um mecanismo genérico de iteração (também chamado **de looping)** é mostrado na caixa da Figura 2-6. Como uma declaração condicional, começa com um teste. Se o teste for avaliado como True, o programa executará o corpo **do loop** uma vez e voltará para reavaliar o teste. Esse processo é repetido até que o teste seja avaliado como Falso, após o que o controle passa para o código que segue a instrução de iteração.

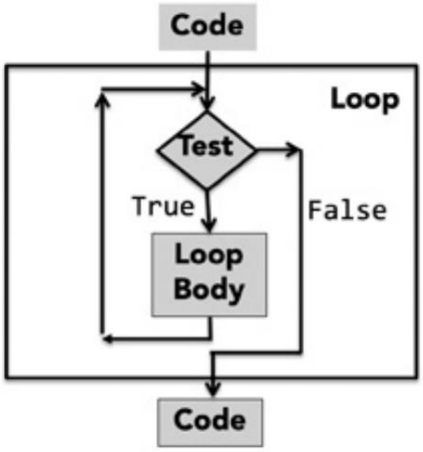

Figura 2-6 Fluxograma para iteração

Podemos escrever o tipo de loop representado na Figura 2-6 usando uma instrução **while .** Considere o código na Figura 2-7.

```
x = 3
ans = 0
num_iterations = 0
while (num_iterations < x):
    ans = ans + x
    num_iterations = num_iterations + 1
print(f'{x}*{x} = {ans}')
```

Figura 2-7 Quadrando um número inteiro, da maneira mais difícil

O código começa vinculando a variável x ao inteiro 3. Em seguida, procede ao quadrado de x usando adição repetitiva. A tabela na Figura 2-8 mostra o valor associado a cada variável cada vez que o teste no início do loop é alcançado. Construímos a tabela manualmente **simulando** o código, ou seja, fingimos ser um interpretador Python e executamos o programa usando lápis e papel. Usar lápis e papel pode parecer estranho, mas é uma excelente maneira de entender como um programa se comporta.19

Test #	x	ans	num_iterations
1	3	0	0
2	3	3	1
3	3	6	2
4	3	9	3

Figura 2-8 Simulação manual de um pequeno programa

Na quarta vez que o teste é alcançado, ele é avaliado como Falso e o fluxo de controle prossegue para a instrução de impressão após o loop. Para quais valores de x esse programa terminará? Há três casos a serem considerados: x == 0, x > 0 e x < 0.

Suponha que x == 0. O valor inicial de num_iterations também será 0 e o corpo do loop nunca será executado.

Suponha que x > 0. O valor inicial de num_iterations será menor que e o corpo do loop será executado pelo menos uma vez. Cada vez que o corpo do loop é executado, o valor de num_iterations aumenta exatamente em 1. Isso significa que, como num_iterations começou antes de um número finito de iterações do loop, num_iterations será igual a x. Nesse ponto, o teste de loop é avaliado como Falso e o controle prossegue para o código que segue a instrução while .

Suponha que x < 0. Algo muito ruim acontece. O controle entrará no loop e cada iteração moverá num_iterations para mais longe de x do que para mais perto dele. O programa, portanto, continuará executando o loop indefinidamente (ou até que algo ruim ocorra, por exemplo, um erro de estouro). Como poderíamos remover essa falha no

programa? Alterar o teste para num_iterations < abs(x) quase funciona. O loop termina, mas imprime um valor negativo. Se a instrução de atribuição dentro do loop também for alterada, para ans = ans + abs(x), o código funcionará corretamente.

Exercício de dedo: Substitua o comentário no código a seguir por um loop while .

```
num_x = int(input('Quantas vezes devo imprimir a letra X? ')) to_print = #concatenar X
para
                    "
to_print num_x
vezes print(to_print)
```

Às vezes é conveniente sair de um loop sem testar a condição do loop. A execução de uma instrução **break** encerra o loop no qual ela está contida e transfere o controle para o código imediatamente após o loop. Por exemplo, o código

```
#Encontre um inteiro positivo divisível por 11 e 12 x = 1

enquanto verdadeiro:
        se x%11 == 0 e x%12 == 0:
                quebrar
        x = x + 1
print(x, 'é divisível por 11 e 12')
```

estampas

```
132 é divisível por 11 e 12
```

Se uma instrução break for executada dentro de um loop aninhado (um loop dentro de outro loop), a quebra encerrará o loop interno.

Exercício de dedo: Escreva um programa que peça ao usuário para inserir 10 números inteiros e, em seguida, imprima o maior número ímpar que foi inserido. Se nenhum número ímpar foi inserido, ele deve imprimir uma mensagem para esse efeito.

2.6 Para loops e alcance

Os loops while que usamos até agora são altamente estilizados, muitas vezes iterando sobre uma sequência de inteiros. Python fornece uma linguagem

mecanismo, o loop **for ,** que pode ser usado para simplificar programas contendo esse tipo de iteração.

A forma geral de uma instrução for é (lembre-se de que as palavras em itálico são descrições do que pode aparecer, não código real):

```
para variável em sequência: bloco de
        código
```

A variável seguinte for é vinculada ao primeiro valor na sequência e o bloco de código é executado. A variável recebe então o segundo valor na sequência e o bloco de código é executado novamente.
O processo continua até que a sequência se esgote ou uma instrução break seja executada dentro do bloco de código. Por exemplo, o código

```
total = 0 para
num in (77, 11, 3): total = total +
      num print(total)
```

imprimirá 91. A expressão (77, 11, 3) é uma **tupla.** Discutimos as tuplas em detalhes na Seção 5. Por enquanto, pense apenas em uma tupla como uma sequência de valores.

A sequência de valores vinculados à ***variável*** é mais comumente gerada usando a função interna que retorna uma série de **intervalos** inteiros. A função e de
três argumentos inteiros: start, stop e step. Produz a progressão start, start + step, faixa
start + 2*step, etc. Se step for positivo, o último elemento é o maior inteiro tal que (start + i*step) é estritamente menor que stop. Se step for negativo, o último elemento é o menor inteiro tal que (start + i*step) é maior que stop. Por exemplo, a expressão range(5, 40, 10) produz a sequência 5, 15, 25, 35, e a expressão range(40, 5, -10) produz a sequência 40, 30, 20, 10.

Se o primeiro argumento para range for omitido, o padrão será 0, e se o último argumento (o tamanho do passo) for omitido, o padrão será 1. Por exemplo, range(0, 3) e range(3) ambos produzem a sequência 0, 1, 2. Os números na progressão são gerados “conforme necessário”, portanto, mesmo expressões como range(1000000) consomem pouca memória. Discutiremos com mais profundidade na Seção 5.2.

faixa

Considere o código

```
x = 4
para i no intervalo(x): print(i)
```

imprime

```
0
1
2
3
```

O código da Figura 2-9 reimplementa o algoritmo da Figura 2-7 para elevar ao quadrado um inteiro (corrigido para que funcione com números negativos). Observe que, ao contrário da implementação do loop while , o número de iterações não é controlado por um teste explícito e a variável de índice num_iterations não é incrementada explicitamente.

```
x = 3
xans = 0
for num_iterations in range(abs(x)):
    ans = ans + abs(x)
print(f'{x}*{x} = {ans}')
```

Figura 2-9 Usando uma instrução for

Observe que o código na Figura 2-9 não altera o valor de num_iterations dentro do corpo do loop for . Isso é típico, mas não necessário, o que levanta a questão do que acontece se a variável de índice for modificada dentro do loop for . Considerar

```
para i no intervalo(2): print(i)
        i = 0 print(i)
```

Você acha que ele imprimirá 0, 0, 1, 0 e depois parará? Ou faça você acha que vai imprimir 0 repetidamente?

A resposta é 0, 0, 1, 0. Antes da primeira iteração do loop for , a função é avaliada e o primeiro valor na sequência que ela produz é atribuído à variável de índice, i. No começo de

a cada iteração subsequente do loop, i recebe o próximo valor na sequência. Quando a sequência é esgotada, o loop termina.
O loop for acima é equivalente ao código

```
índice = 0
last_index = 1 while
index <= last_index: i = index print(i) i =
    0 print(i) index
    = index + 1
```

Observe, a propósito, que o código com o loop while é consideravelmente mais complicado do que o loop for . O loop for é um mecanismo linguístico conveniente.

Agora, o que você acha

```
x = 1
para i no intervalo(x):
    imprimir(i)
    x = 4
```

impressões? Apenas 0, porque os argumentos para a função na linha com for são avaliados logo antes da primeira iteração do loop e não são reavaliados nas iterações subsequentes.

fixa

Agora, vamos ver com que frequência as coisas são avaliadas quando aninhamos loops. Considerar

```
x = 4
para j no intervalo (x): para
    i no intervalo (x): x = 2
```

Quantas vezes cada um dos dois loops é executado? Já vimos que o range(x) que controla o loop externo é avaliado na primeira vez que é atingido e não reavaliado a cada iteração, portanto, são quatro iterações do loop externo. Isso implica que o loop for interno é alcançado quatro vezes. A primeira vez que é atingida, a variável x = 4, então haverá quatro iterações. No entanto, nas próximas três vezes que for atingido, x = 2, haverá duas iterações de cada vez.

Conseqüentemente, se você executar

```
x = 3
for j in range(x):
        print('Iteração do loop externo') for i in range(x):
        print(' x = 2
                                    Iteração do loop interno')
```

ele imprime

```
Iteração do loop externo
        Iteração do loop interno
        Iteração do loop interno
        Iteração do loop interno
Iteração do loop externo
        Iteração do loop interno
        Iteração do loop interno
Iteração do loop externo
        Iteração do loop interno
        Iteração do loop interno
```

A instrução for pode ser usada em conjunto com o **operador in** para iterar convenientemente os caracteres de uma string. Por exemplo,

```
total = 0 para
c em '12345678': total = total
        + int(c) print(total)
```

soma os dígitos na string denotada pelo literal '12345678' e imprime o total.

Exercício de dedo: Escreva um programa que imprima a soma dos números primos maiores que 2 e menores que 1000. Dica: você provavelmente deseja usar um loop for que é um teste de primalidade aninhado dentro de um loop for que itera sobre os inteiros ímpares entre 3 e 999.

2.7 Questões de estilo

Grande parte deste livro é dedicada a ajudá-lo a aprender uma linguagem de programação. Mas saber um idioma e saber usar bem um idioma são duas coisas diferentes. Considere os dois seguintes
frases:

"Todo mundo sabe que se um homem é solteiro e tem muitos dinheiro, ele precisa se casar.

"É uma verdade universalmente reconhecida que um único homem em possuidor de uma boa fortuna, deve estar precisando de uma esposa."20

Cada uma é uma sentença adequada em inglês e cada uma significa aproximadamente a mesma coisa. Mas eles não são igualmente atraentes e talvez não sejam igualmente fáceis de entender. Assim como o estilo é importante ao escrever em inglês, o estilo é importante ao escrever em Python.
No entanto, embora ter uma voz distinta possa ser um trunfo para um romancista, não é um trunfo para um programador. Quanto menos tempo os leitores de um programa tiverem para pensar em coisas irrelevantes para o significado do código, melhor. É por isso que bons programadores seguem convenções de codificação projetadas para tornar os programas fáceis de entender, em vez de divertidos de ler.

A maioria dos programadores Python segue as convenções estabelecidas no **guia de estilo PEP 8.** 21 Como todos os conjuntos de convenções, algumas de suas prescrições são arbitrárias. Por exemplo, ele prescreve o uso de quatro espaços para recuos. Por que quatro espaços e não três ou cinco? Nenhuma razão particularmente boa. Mas se todos usarem o mesmo número de espaços, é mais fácil ler (e talvez combinar) códigos escritos por pessoas diferentes. De modo mais geral, se todos usarem o mesmo conjunto de convenções ao escrever Python, os leitores poderão se concentrar em entender a semântica do código em vez de desperdiçar ciclos mentais assimilando decisões estilísticas.

As convenções mais importantes têm a ver com a nomenclatura. Já discutimos a importância de usar nomes de variáveis que transmitam o significado da variável. Frases nominais funcionam bem para isso.
Por exemplo, usamos o nome num_iterations para uma variável que denota o número de iterações. Quando um nome inclui várias palavras, a convenção em Python é usar um sublinhado (_) para separar as palavras. Novamente, esta convenção é arbitrária. Alguns programadores preferem usar o que geralmente é chamado de camelCase, por exemplo, numIterations — argumentando que é mais rápido digitar e usa menos espaço.

Existem também algumas convenções para nomes de variáveis de um único caractere. O mais importante é evitar o uso de L minúsculo ou I maiúsculo (que são facilmente confundidos com o número um) ou O maiúsculo (que é facilmente confundido com o número zero).

Chega de convenções por enquanto. Voltaremos ao tópico conforme apresentamos vários aspectos do Python.

Agora cobrimos praticamente tudo sobre Python que você precisa saber para começar a escrever programas interessantes que lidam com números e strings. No próximo capítulo, faremos uma pequena pausa no aprendizado de Python e usaremos o que você já aprendeu para resolver alguns problemas simples.

2.8 Termos Introduzidos no Capítulo

linguagem de baixo

nível linguagem de alto

nível linguagem

interpretada linguagem

compilada código-fonte

Código da máquina

Ambiente

de desenvolvimento integrado Python (IDE)

anaconda

Spyder

shell do console

IPython

programa (script)

comando (declaração)

objeto

tipo

objeto escalar

objeto não escalar

literal

bool de ponto
flutuante

Nenhum

valor da

expressão do
operador

variável de
prompt do shell

palavra

reservada de
atribuição vinculativa

comentário (no código)

programa de linha reta

programa de ramificação
condicional

indentação (em Python)
instrução aninhada

expressão composta
tempo constante

complexidade computacional

strings de expressão

condicional sobrecarregado

operador repetição tipo

de operador

verificação

indexação corte

conversão de tipo (casting)

entrada de expressão de string

formatada

Unicode

pseudocódigo de

iteração

(looping)

durante a simulação de loop manual

quebrar

for loop

tupla

faixa

no operador

PEP 8 guia de estilo

---

9 Alegadamente, o nome Python foi escolhido como uma homenagem à trupe de comédia britânica Monty Python. Isso leva a pensar que o nome IDLE é um trocadilho com Eric Idle, um membro da trupe.

10 Como o Anaconda é atualizado com frequência, quando você ler isto, a aparência da janela pode ter mudado.

11 Se você não gostar da aparência da janela do Spyder , clique na chave inglesa na barra de ferramentas para abrir a janela Preferences e altere as configurações como achar melhor.

12 Funções são discutidas na Seção 4.

13 Sim, os átomos não são verdadeiramente indivisíveis. No entanto, separá-los não é fácil e isso pode ter consequências nem sempre desejáveis.

14 Se você acredita que o valor real de ÿ não é 3, você está certo. Nós até demonstramos esse fato na Seção 18.4.

15 “O que há em um nome? Aquilo que chamamos de rosa por qualquer outro nome cheiraria tão doce.”

16 Ao contrário de muitas linguagens de programação, Python não tem tipo correspondente a um personagem. Em vez disso, ele usa strings de comprimento 1.

17 Esses modificadores são os mesmos modificadores usados no formato .format método associado a strings.

18 Em 2016, mais de 85% das páginas da web foram codificadas usando UTF-8.

19 Também é possível simular manualmente um programa usando caneta e papel, ou mesmo um editor de texto.

20 ***Orgulho e Preconceito,*** Jane Austen.

21 PEP é um acrônimo que significa “Python Enhancement Proposta." PEP 8 foi escrito em 2001 por Guido van Rossum, Barry Warsaw e Nick Coghlan.