2
Variáveis e tipos de dados simples

Neste capítulo conheceremos os diferentes tipos de dados com os quais podemos trabalhar em nossos programas Python. Aprenderemos também a armazenar dados em variáveis e usar essas variáveis em nossos programas.

O que realmente acontece quando executamos hello_world.py

Vamos observar com mais detalhes o que Python faz quando executamos *hello_world.py*. O fato é que Python realiza muitas tarefas, mesmo quando executa um programa simples: hello_world.py print("Hello Python world!") Quando executar esse código, você deverá ver a saída a seguir: Hello Python world!

Ao executar *hello_world.py*, o arquivo terminado em *.py* indica que é um programa Python. Seu editor então passa o programa pelo *interpretador Python*, que lê o programa todo e determina o que cada palavra significa. Por exemplo, quando vê a palavra `print`, o interpretador exibe na tela o que quer que esteja dentro dos parênteses.

À medida que você escrever seus programas, o editor dará destaque a diferentes partes do código de modo distinto. Por exemplo, ele reconhece que `print` é o nome de uma função e exibe essa palavra em azul. Reconhece que "Hello Python world!" não é um código Python e exibe essa frase em laranja. Esse recurso se chama *destaque de sintaxe* e é muito útil quando você começar a escrever seus próprios programas.

Variáveis

Vamos experimentar usar uma variável em *hello_world.py*. Acrescente uma nova linha no início do arquivo e modifique a segunda linha: message = "Hello Python world!"

```
print(message) Execute esse programa para ver o que acontece. Você deverá
ver a mesma saída vista anteriormente: Hello Python world!
```

Acrescentamos uma *variável* chamada `message`. Toda variável armazena um *valor*, que é a informação associada a essa variável. Nesse caso, o

valor é o texto “Hello Python world!”.

Acrescentar uma variável implica um pouco mais de trabalho para o interpretador Python. Quando processa a primeira linha, ele associa o texto “Hello Python world!” à variável `message`. Quando chega à segunda linha, o interpretador exibe o valor associado à `message` na tela.

Vamos expandir esse programa modificando *hello_world.py* para que exiba uma segunda mensagem. Acrescente uma linha em branco em *hello_world.py* e, em seguida, adicione duas novas linhas de código:

```
message = "Hello Python world!"
  print(message)
  message = "Hello Python Crash Course world!"
  print(message) Ao executar hello_world.py agora você deverá ver duas
  linhas na saída: Hello Python world!

  Hello Python Crash Course world!
```

Podemos mudar o valor de uma variável em nosso programa a qualquer momento, e Python sempre manterá o controle do valor atual.

Nomeando e usando variáveis

Ao usar variáveis em Python, é preciso seguir algumas regras e diretrizes. Quebrar algumas dessas regras provocará erros; outras diretrizes simplesmente ajudam a escrever um código mais fácil de ler e de entender. Lembre-se de ter as seguintes regras em mente: • Nomes de variáveis podem conter apenas letras, números e underscores. Podem começar com uma letra ou um underscore, mas não com um número. Por exemplo, podemos chamar uma variável de *message_1*, mas não de *1_message*.

- Espaços não são permitidos em nomes de variáveis, mas underscores podem ser usados para separar palavras em nomes de variáveis. Por exemplo, *greeting_message* funciona, mas *greeting message* causará erros.
- Evite usar palavras reservadas e nomes de funções em Python como nomes de variáveis, ou seja, não use palavras que Python reservou para um propósito particular de programação, por exemplo, a palavra `print`. (Veja a seção “Palavras reservadas e funções embutidas de Python”.) • Nomes de variáveis devem ser concisos, porém descritivos. Por exemplo, *name* é melhor que *n*, *student_name*

é melhor que *s_n* e *name_length* é melhor que *length_of_persons_name*.

- Tome cuidado ao usar a letra minúscula *l* e a letra maiúscula *O*, pois elas podem ser confundidas com os números *1* e *0*.

Aprender a criar bons nomes de variáveis, em especial quando seus programas se tornarem mais interessantes e complicados, pode exigir um pouco de prática. À medida que escrever mais programas e começar a ler códigos de outras pessoas, você se tornará mais habilidoso na criação de nomes significativos.

NOTA As variáveis Python que você está usando no momento devem utilizar letras minúsculas. Você não terá erros se usar letras maiúsculas, mas é uma boa ideia evitá-las por enquanto.

Evitando erros em nomes ao usar variáveis

Todo programador comete erros, e a maioria comete erros todos os dias. Embora bons programadores possam criar erros, eles também sabem responder a esses erros de modo eficiente. Vamos observar um erro que você provavelmente cometerá no início e vamos aprender a corrigi-lo.

Escreveremos um código que gera um erro propositadamente. Digite o código a seguir, incluindo a palavra *mesage* com um erro de ortografia, conforme mostrada em negrito: message = "Hello Python Crash Course reader!"

```
print(mesage) Quando houver um erro em seu programa, o interpretador
Python faz o melhor que puder para ajudar você a descobrir onde está o
problema. O interpretador oferece um traceback quando um programa não é
capaz de executar com sucesso. Um traceback é um registro do ponto em que
o interpretador se deparou com problemas quando tentou executar seu
código. Eis um exemplo do traceback fornecido por Python após o nome da
variável ter sido digitado incorretamente por engano: Traceback (most
recent call last): ❶  File "hello_world.py", line 2, in <module> ❷
print(mesage) ❸ NameError: name 'mesage' is not defined A saída em ❶
informa que um erro ocorreu na linha 2 do arquivo hello_world.py. O
interpretador mostra essa linha para nos ajudar a identificar o erro
rapidamente ❷ e informa o tipo do erro encontrado ❸. Nesse caso, ele
encontrou um erro de nome e informa que a variável exibida, mesage, não
foi definida. Python não é capaz de identificar o nome da variável
especificada. Um erro de nome geralmente quer dizer que esquecemos de
definir o valor de uma variável antes de usá-la ou que cometemos um erro
de ortografia quando fornecemos o nome da variável.
```

É claro que, nesse exemplo, omitimos a letra *s* do nome da variável `message` na segunda linha. O interpretador Python não faz uma verificação de ortografia em seu código, mas garante que os nomes das variáveis estejam escritos de forma consistente. Por exemplo, observe o que acontece quando escrevemos *message* incorretamente em outro ponto do código também: mesage = "Hello Python Crash Course reader!"

```
print(mesage) Nesse caso, o programa executa com sucesso!

Hello Python Crash Course reader!
```

Os computadores são exigentes, mas não se preocupam com uma ortografia correta ou incorreta. Como resultado, você não precisa levar em consideração a ortografia ou as regras gramaticais do inglês (ou de português) quando tentar criar nomes de variáveis e escrever código.

Muitos erros de programação são apenas erros de digitação de um caractere em uma linha de um programa. Se você gasta muito tempo procurando um desses erros, saiba que está em boa companhia. Muitos programadores experientes e talentosos passam horam caçando esses tipos de pequenos erros. Tente rir da situação e siga em frente, sabendo que isso acontecerá com frequência durante sua vida de programador.

NOTA A melhor maneira de entender novos conceitos de programação é tentar usá-los em seus programas. Se não souber o que fazer quando estiver trabalhando em um exercício deste livro, procure dedicar-se a outra tarefa por um tempo. Se continuar sem conseguir avançar, analise a parte relevante do capítulo. Se ainda precisar de ajuda, veja as sugestões no Apêndice C.

FAÇA VOCÊ MESMO

Escreva um programa separado para resolver cada um destes exercícios. Salve cada programa com um nome de arquivo que siga as convenções-padrão de Python, com letras minúsculas e underscores, por exemplo, *simple_message.py* e *simple_messages.py*.

2.1 – Mensagem simples: Armazene uma mensagem em uma variável e, em seguida, exiba essa mensagem.

2.2 – Mensagens simples: Armazene uma mensagem em uma variável e, em seguida, exiba essa mensagem. Então altere o valor de sua variável para uma nova mensagem e mostre essa nova mensagem.

Strings

Como a maior parte dos programas define e reúne algum tipo de dado e então faz algo útil com eles, classificar diferentes tipos de dados é conveniente. O primeiro tipo de dado que veremos é a string. As strings, à primeira vista, são bem simples, mas você pode usá-las de vários modos diferentes.

Uma *string* é simplesmente uma série de caracteres. Tudo que estiver entre aspas é considerada uma string em Python, e você pode usar aspas simples ou duplas em torno de suas strings, assim: "This is a string."

```
'This is also a string.'
```

Essa flexibilidade permite usar aspas e apóstrofos em suas strings: 'I told my friend, "Python is my favorite language!"'

```
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Vamos explorar algumas das maneiras de usar strings.

Mudando para letras maiúsculas e minúsculas em uma string usando métodos

Uma das tarefas mais simples que podemos fazer com strings é mudar o tipo de letra, isto é, minúscula ou maiúscula, das palavras de uma string. Observe o código a seguir e tente determinar o que está acontecendo: name.py name = "ada lovelace"

```
print(name.title()) Salve esse arquivo como name.py e então execute-o.
Você deverá ver esta saída: Ada Lovelace Nesse exemplo, a string com
letras minúsculas "ada lovelace" é armazenada na variável name. O método
title() aparece depois da variável na instrução print(). Um método é uma
ação que Python pode executar em um dado. O ponto (.) após name em
name.title() informa a Python que o método title() deve atuar na variável
name. Todo método é seguido de um conjunto de parênteses, pois os
métodos, com frequência, precisam de informações adicionais para realizar
sua tarefa. Essas informações são fornecidas entre os parênteses. A
função title() não precisa de nenhuma informação adicional, portanto seus
parênteses estão vazios.
```

`title()` exibe cada palavra com uma letra maiúscula no início. Isso é útil, pois, muitas vezes, você vai querer pensar em um nome como uma informação. Por exemplo, você pode querer que seu programa reconheça os valores de entrada `Ada`, `ADA` e `ada` como o mesmo nome e

exiba todos eles como Ada.

Vários outros métodos úteis estão disponíveis para tratar letras maiúsculas e minúsculas também. Por exemplo, você pode mudar uma string para que tenha somente letras maiúsculas ou somente letras minúsculas, assim: name = "Ada Lovelace"

```
print(name.upper()) print(name.lower()) Essas instruções exibirão o
seguinte: ADA LOVELACE
ada lovelace O método lower() é particularmente útil para armazenar
dados. Muitas vezes, você não vai querer confiar no fato de seus usuários
fornecerem letras maiúsculas ou minúsculas, portanto fará a conversão das
strings para letras minúsculas antes de armazená-las. Então, quando
quiser exibir a informação, usará o tipo de letra que fizer mais sentido
para cada string.
```

Combinando ou concatenando strings

Muitas vezes, será conveniente combinar strings. Por exemplo, você pode querer armazenar um primeiro nome e um sobrenome em variáveis separadas e, então, combiná-las quando quiser exibir o nome completo de alguém: first_name = "ada"

```
last_name = "lovelace"
❶ full_name = first_name + " " + last_name
print(full_name) Python usa o símbolo de adição (+) para combinar
strings. Nesse exemplo, usamos + para criar um nome completo combinando
first_name, um espaço e last_name ❶, o que resultou em: ada lovelace
Esse método de combinar strings se chama concatenação. Podemos usar
concatenação para compor mensagens completas usando informações
armazenadas em uma variável. Vamos observar um exemplo: first_name =
"ada"

last_name = "lovelace"
full_name = first_name + " " + last_name
❶ print("Hello, " + full_name.title() + "!") Nesse caso, o nome completo é
usado em ❶, em uma sentença que saúda o usuário, e o método title() é
utilizado para formatar o nome de forma apropriada. Esse código devolve uma
saudação simples, porém formatada de modo elegante: Hello, Ada Lovelace!
```

Podemos usar concatenação para compor uma mensagem e então armazenar a mensagem completa em uma variável: first_name = "ada"

```
last_name = "lovelace"
full_name = first_name + " " + last_name
❶ message = "Hello, " + full_name.title() + "!"
```

❷ print(message) Esse código também exibe a mensagem "Hello, Ada Lovelace!", mas armazenar a mensagem em uma variável em ❶ deixa a instrução print final em ❷ muito mais simples.

Acrescentando espaços em branco em strings com tabulações ou quebras de linha

Em programação, *espaços em branco* se referem a qualquer caractere que não é mostrado, como espaços, tabulações e símbolos de fim de linha. Podemos usar espaços em branco para organizar a saída de modo que ela seja mais legível aos usuários.

Para acrescentar uma tabulação em seu texto, utilize a combinação de caracteres \t como mostrado em ❶: >>> **print("Python")** Python
❶ >>> **print("\tPython")** Python Para acrescentar uma quebra de linha em uma string, utilize a combinação de caracteres \n: >>>
print("Languages:\nPython\nC\nJavaScript") Languages: Python
C
JavaScript Também podemos combinar tabulações e quebras de linha em uma única string. A string "\n\t" diz a Python para passar para uma nova linha e iniciar a próxima com uma tabulação. O exemplo a seguir mostra como usar uma string de uma só linha para gerar quatro linhas na saída:
>>> **print("Languages:\n\tPython\n\tC\n\tJavaScript")** Languages: Python C

JavaScript Quebras de linha e tabulações serão muito úteis nos próximos dois capítulos, quando começaremos a gerar muitas linhas de saída usando apenas algumas linhas de código.

Removendo espaços em branco

Espaços em branco extras podem ser confusos em seus programas. Para os programadores, 'python' e 'python ' parecem praticamente iguais. Contudo, para um programa, são duas strings diferentes. Python identifica o espaço extra em 'python ' e o considera significativo, a menos que você informe o contrário.

É importante pensar em espaços em branco, pois, com frequência, você vai querer comparar duas strings para determinar se são iguais. Por exemplo, uma situação importante pode envolver a verificação dos nomes de usuário das pessoas quando elas fizerem login em um site. Espaços em branco extras podem ser confusos em situações muito mais simples também. Felizmente, Python facilita eliminar espaços em branco extras dos dados fornecidos pelas pessoas.

Python é capaz de encontrar espaços em branco dos lados direito e esquerdo de uma string. Para garantir que não haja espaços em branco do lado direito de uma string, utilize o método `rstrip()`.

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language 'python '
❸ >>> favorite_language.rstrip() 'python'
❹ >>> favorite_language 'python '
```

O valor armazenado em `favorite_language` em ❶ contém um espaço em branco extra no final da string. Quando solicitamos esse valor a Python em uma sessão de terminal, podemos ver o espaço no final do valor ❷. Quando o método `rstrip()` atua na variável `favorite_language` em ❸, esse espaço extra é removido. Entretanto, a remoção é temporária. Se solicitar o valor de `favorite_language` novamente, você poderá ver que a string é a mesma que foi fornecida, incluindo o espaço em branco extra ❹.

Para remover o espaço em branco da string de modo permanente, você deve armazenar o valor com o caractere removido de volta na variável: >>> **favorite_language = 'python '**

```
❶ >>> favorite_language = favorite_language.rstrip() >>> favorite_language
'python'
```

Para remover o espaço em branco da string, você deve remover o espaço em branco do lado direito da string e então armazenar esse valor de volta na variável original, como mostrado em ❶. Alterar o valor de uma variável e então armazenar o novo valor de volta na variável original é uma operação frequente em programação. É assim que o valor de uma variável pode mudar à medida que um programa é executado ou em resposta à entrada de usuário.

Também podemos remover espaços em branco do lado esquerdo de uma string usando o método `lstrip()`, ou remover espaços em branco dos dois lados ao mesmo tempo com `strip()`: ❶ >>> **favorite_language = ' python '**

```
❷ >>> favorite_language.rstrip() ' python'
❸ >>> favorite_language.lstrip() 'python '
❹ >>> favorite_language.strip() 'python'
```

Nesse exemplo, começamos com um valor que tem espaços em branco no início e no fim ❶. Então removemos os espaços extras do lado direito em ❷, do lado esquerdo em ❸ e de ambos os lados em ❹.

Fazer experimentos com essas funções de remoção pode ajudar você a ter familiaridade com a manipulação de strings. No mundo real, essas funções de remoção são usadas com mais frequência para limpar entradas de usuário antes de armazená-las em um programa.

Evitando erros de sintaxe com strings

Um tipo de erro que você poderá ver com certa regularidade é um erro de sintaxe. Um *erro de sintaxe* ocorre quando Python não reconhece uma seção de seu programa como um código Python válido. Por exemplo, se usar um apóstrofo entre aspas simples, você produzirá um erro. Isso acontece porque Python interpreta tudo que estiver entre a primeira aspa simples e o apóstrofo como uma string. Ele então tenta interpretar o restante do texto como código Python, o que causa erros.

Eis o modo de usar aspas simples e duplas corretamente. Salve este programa como *apostrophe.py* e então execute-o: apostrophe.py message = "One of Python's strengths is its diverse community."

```
print(message) O apóstrofo aparece entre um conjunto de aspas duplas,
portanto o interpretador Python não terá nenhum problema para ler a
string corretamente: One of Python's strengths is its diverse community.
```

No entanto, se você usar aspas simples, o interpretador Python não será capaz de identificar em que ponto a string deve terminar: message = 'One of Python's strengths is its diverse community.'

```
print(message) Você verá a saída a seguir: File "apostrophe.py", line 1

  message = 'One of Python's strengths is its diverse community.'
  ^❶ SyntaxError: invalid syntax Na saída, podemos ver que o erro ocorre
em ❶, logo depois da segunda aspa simples. Esse erro de sintaxe informa
que o interpretador não reconheceu algo como código Python válido. Os
erros podem ter origens variadas, e destacarei alguns erros comuns à
medida que surgirem. Você poderá ver erros de sintaxe com frequência
enquanto aprende a escrever código Python apropriado. Erros de sintaxe
também são o tipo menos específico de erro, portanto podem ser difíceis e
frustrantes para identificar e corrigir. Se você não souber o que fazer
quanto a um erro particularmente persistente, consulte as sugestões no
Apêndice C.
```

NOTA O recurso de destaque de sintaxe de seu editor deve ajudar você a identificar alguns erros de sintaxe rapidamente quando escrever seus programas. Se você vir código Python em destaque como se fosse inglês, ou inglês destacado como se fosse código Python, é provável que haja uma aspa sem correspondente em algum ponto de seu arquivo.

Exibindo informações em Python 2

A instrução `print` tem uma sintaxe levemente diferente em Python 2: >>> **`python2.7`**

```
>>> print "Hello Python 2.7 world!"
Hello Python 2.7 world!
```

Os parênteses não são necessários em torno de frases que você quer exibir em Python 2. Tecnicamente, `print` é uma função em Python 3, motivo pelo qual os parênteses são necessários. Algumas instruções `print` em Python 2 incluem parênteses, mas o comportamento pode ser um pouco diferente do que você verá em Python 3. Basicamente, quando estiver vendo código escrito em Python 2, espere ver algumas instruções `print` com parênteses e outras sem.

FAÇA VOCÊ MESMO

Salve cada um dos exercícios a seguir em um arquivo separado com um nome como *name_cases.py*. Se não souber o que fazer, descase um pouco ou consulte as sugestões que estão no Apêndice C.

2.3 – Mensagem pessoal: Armazene o nome de uma pessoa em uma variável e apresente uma mensagem a essa pessoa. Sua mensagem deve ser simples, como "Alô Eric, você gostaria de aprender um pouco de Python hoje?".

2.4 – Letras maiúsculas e minúsculas em nomes: Armazene o nome de uma pessoa em uma variável e então apresente o nome dessa pessoa em letras minúsculas, em letras maiúsculas e somente com a primeira letra maiúscula.

2.5 – Citação famosa: Encontre uma citação de uma pessoa famosa que você admire. Exiba a citação e o nome do autor. Sua saída deverá ter a aparência a seguir, incluindo as aspas: Albert Einstein certa vez disse: "Uma pessoa que nunca cometeu um erro jamais tentou nada novo."

2.6 – Citação famosa 2: Repita o Exercício 2.5, porém, desta vez, armazene o nome da pessoa famosa em uma variável chamada `famous_person`. Em seguida, componha sua mensagem e armazene-a em uma nova variável chamada `message`. Exiba sua mensagem.

2.7 – Removendo caracteres em branco de nomes: Armazene o nome de uma pessoa e inclua alguns caracteres em branco no início e no final do nome. Lembre-se de usar cada combinação de caracteres, `"\t"` e `"\n"`, pelo menos uma vez.

Exiba o nome uma vez, de modo que os espaços em branco em torno do nome sejam mostrados. Em seguida, exiba o nome usando cada uma das três funções de remoção de espaços: `lstrip()`, `rstrip()` e `strip()`.

Números

Os números são usados com muita frequência em programação para armazenar pontuações em jogos, representar dados em visualizações, guardar informações em aplicações web e assim por diante. Python trata números de várias maneiras diferentes, de acordo com o modo como são usados. Vamos analisar inicialmente como Python trata inteiros, pois eles são os dados mais simples para trabalhar.

Inteiros

Você pode somar (+), subtrair (-), multiplicar (*) e dividir (/) inteiros em Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Em uma sessão de terminal, Python simplesmente devolve o resultado da operação. Dois símbolos de multiplicação são usados em Python para representar exponenciais: >>> **3 ** 2**

```
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

A linguagem Python também aceita a ordem das operações, portanto você pode fazer várias operações em uma expressão. Também podemos usar parênteses para modificar a ordem das operações para que Python possa avaliar sua expressão na ordem que você especificar. Por exemplo: >>> **2 + 3*4**

```
14
>>> (2 + 3) * 4
20
```

Os espaços nesses exemplos não têm nenhum efeito no modo como Python avalia as expressões: eles simplesmente ajudam a identificar mais

rapidamente as operações que têm prioridade quando lemos o código.

Números de ponto flutuante

Python chama qualquer número com um ponto decimal de *número de ponto flutuante* (float). Esse termo é usado na maioria das linguagens de programação e refere-se ao fato de um ponto decimal poder aparecer em qualquer posição em um número. Toda linguagem de programação deve ser cuidadosamente projetada para lidar de modo adequado com números decimais de modo que eles se comportem de forma apropriada, não importa em que lugar o ponto decimal apareça.

Na maioria das ocasiões, podemos usar decimais sem nos preocupar com o modo como eles se comportam. Basta fornecer os números que você quer usar, e Python provavelmente fará o que você espera: >>> **0.1 + 0.1**

```
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

No entanto, tome cuidado, pois, às vezes, você poderá obter um número arbitrário de casas decimais em sua resposta: >>> **0.2 + 0.1**

```
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Isso acontece em todas as linguagens e não é motivo para muita preocupação. Python tenta encontrar uma forma de representar o resultado do modo mais exato possível, o que, às vezes, é difícil, considerando a maneira como os computadores devem representar os números internamente. Basta ignorar as casas decimais extras por enquanto; você aprenderá a lidar com casas extras quando for necessário nos projetos da Parte II.

Evitando erros de tipo com a função str()

Com frequência, você vai querer usar o valor de uma variável em uma mensagem. Por exemplo, suponha que você queira desejar feliz

aniversário a alguém. Você poderia escrever um código como este: birthday.py age = 23

```
message = "Happy " + age + "rd Birthday!"


print(message) Você esperaria que esse código exibisse a seguinte
saudação simples de feliz aniversário: Happy 23rd birthday!. Contudo, se
executar esse código, verá que ele gera um erro: Traceback (most recent
call last): File "birthday.py", line 2, in <module> message = "Happy " +
age + "rd Birthday!"
```

❶ TypeError: Can't convert 'int' object to str implicitly É um *erro de tipo*. Significa que Python não é capaz de reconhecer o tipo de informação que você está usando. Nesse exemplo, Python vê que você está usando uma variável em ❶ cujo valor é um inteiro (int), mas não tem certeza de como interpretar esse valor. O interpretador sabe que a variável poderia representar um valor numérico 23 ou os caracteres *2* e *3*. Quando usar inteiros em strings desse modo, você precisará especificar explicitamente que quer que Python utilize o inteiro como uma string de caracteres. Podemos fazer isso envolvendo a variável com a função str(); essa função diz a Python para representar valores que não são strings como strings: age = 23

```
message = "Happy " + str(age) + "rd Birthday!"


print(message) Com isso, Python sabe que você quer converter o valor
numérico 23 em uma string e exibir os caracteres 2 e 3 como parte da
mensagem de feliz aniversário. Agora você obterá a mensagem esperada, sem
erros: Happy 23rd Birthday!
```

Trabalhar com números em Python é simples na maior parte do tempo. Se você estiver obtendo resultados inesperados, verifique se Python está interpretando seus números da forma desejada, seja como um valor numérico, seja como um valor de string.

Inteiros em Python 2

Python 2 devolve um resultado um pouco diferente quando dividimos dois inteiros: >>> python2.7

```
>>> 3 / 2
1
```

Em vez de 1.5, Python devolve 1. A divisão de inteiros em Python 2 resulta em um inteiro com o resto truncado. Observe que o resultado não é um inteiro arredondado; o resto é simplesmente omitido.

Para evitar esse comportamento em Python 2, certifique-se de que pelo menos um dos números seja um número de ponto flutuante. Ao fazer isso, o resultado também será um número de ponto flutuante: >>> 3 / 2

```
1
>>> 3.0 / 2
1.5
>>> 3 / 2.0
1.5
>>> 3.0 / 2.0
1.5
```

Esse comportamento da divisão é uma fonte comum de confusão quando as pessoas que estão acostumadas a usar Python 3 começam a usar Python 2 ou vice-versa. Se você usa ou cria código que mistura inteiros e números de ponto flutuante, tome cuidado com comportamentos irregulares.

FAÇA VOCÊ MESMO

2.8 – Número oito: Escreva operações de adição, subtração, multiplicação e divisão que resultem no número 8. Lembre-se de colocar suas operações em instruções `print` para ver os resultados. Você deve criar quatro linhas como esta: print(5 + 3) Sua saída deve simplesmente ser composta de quatro linhas, com o número 8 em cada uma das linhas.

2.9 – Número favorito: Armazene seu número favorito em uma variável. Em seguida, usando essa variável, crie uma mensagem que revele o seu número favorito. Exiba essa mensagem.

Comentários

Comentários são um recurso extremamente útil na maioria das linguagens de programação. Tudo que você escreveu em seus programas até agora é código Python. À medida que seus programas se tornarem mais longos e complicados, você deve acrescentar notas que descrevam a abordagem geral adotada para o problema que você está resolvendo. Um comentário permite escrever notas em seus programas em linguagem natural.

Como escrever comentários?

Em Python, o caractere sustenido (#) indica um comentário. Tudo que vier depois de um caractere sustenido em seu código será ignorado pelo

interpretador Python. Por exemplo: comment.py # Diga olá a todos print("Hello Python people!") Python ignora a primeira linha e executa a segunda.

```
Hello Python people!
```

Que tipos de comentário você deve escrever?

O principal motivo para escrever comentários é explicar o que seu código deve fazer e como você o faz funcionar. Quando estiver no meio do trabalho de um projeto, você entenderá como todas as partes se encaixam. Porém, quando retomar um projeto depois de passar um tempo afastado, é provável que você vá esquecer alguns detalhes. É sempre possível estudar seu código por um tempo e descobrir como os segmentos deveriam funcionar, mas escrever bons comentários pode fazer você economizar tempo ao sintetizar sua abordagem geral em linguagem natural clara.

Se quiser se tornar um programador profissional ou colaborar com outros programadores, escreva comentários significativos. Atualmente, a maior parte dos softwares é escrita de modo colaborativo, seja por um grupo de funcionários em uma empresa, seja por um grupo de pessoas que trabalham juntas em um projeto de código aberto. Programadores habilidosos esperam ver comentários no código, portanto é melhor começar a adicionar comentários descritivos em seus programas agora. Escrever comentários claros e concisos em seu código é um dos hábitos mais benéficos que você pode desenvolver como um novo programador.

Quando estiver decidindo se deve escrever um comentário, pergunte a si mesmo se você precisou considerar várias abordagens antes de definir uma maneira razoável de fazer algo funcionar; em caso afirmativo, escreva um comentário sobre sua solução. É muito mais fácil apagar comentários extras depois que retornar e escrever comentários em um programa pouco comentado. A partir de agora, usarei comentários em exemplos deste livro para ajudar a explicar algumas seções de código.

FAÇA VOCÊ MESMO

2.10 – Acrescentando comentários: Escolha dois dos programas que você escreveu e acrescente pelo menos um comentário em cada um. Se você não tiver nada específico para escrever porque o programa é muito simples no

momento, basta adicionar seu nome e a data de hoje no início de cada arquivo de programa. Em seguida, escreva uma frase que descreva o que o programa faz.

Zen de Python

Durante muito tempo, a linguagem de programação Perl foi o principal sustentáculo da internet. A maioria dos primeiros sites interativos usava scripts Perl. O lema da comunidade Perl na época era: "Há mais de uma maneira de fazer algo". As pessoas gostaram dessa postura por um tempo porque a flexibilidade inscrita na linguagem possibilitava resolver a maior parte dos problemas de várias maneiras. Essa abordagem era aceitável enquanto trabalhávamos em nossos próprios projetos, mas, em algum momento, as pessoas perceberam que a ênfase na flexibilidade dificultava a manutenção de projetos grandes em longo prazo. Revisar código e tentar descobrir o que outra pessoa pensou quando resolvia um problema complexo era difícil, tedioso e consumia bastante tempo.

Programadores Python experientes incentivarão você a evitar a complexidade e buscar a simplicidade sempre que for possível. A filosofia da comunidade Python está contida no "Zen de Python" de Tim Peters. Você pode acessar esse conjunto resumido de princípios para escrever um bom código Python fornecendo `import this` ao seu interpretador. Não reproduzirei todo o "Zen de Python" aqui, mas compartilharei algumas linhas para ajudar a entender por que elas devem ser importantes para você como um programador Python iniciante.

```
>>> import this The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

(**Bonito é melhor que feio**) Os programadores Python adotam a noção de que o código pode ser bonito e elegante. Em programação, as pessoas resolvem problemas. Os programadores sempre respeitaram soluções bem projetadas, eficientes e até mesmo bonitas para os problemas. À medida que conhecer mais a linguagem Python e usá-la para escrever mais códigos, uma pessoa poderá espiar por sobre seu ombro um dia e dizer: "Uau, que código bonito!".

```
Simple is better than complex.
```

(**Simples é melhor que complexo**) Se você puder escolher entre uma

solução simples e outra complexa, e as duas funcionarem, utilize a solução simples. Seu código será mais fácil de manter, e será mais simples para você e para outras pessoas desenvolverem com base nesse código posteriormente.

```
Complex is better than complicated.
```

(**Complexo é melhor que complicado**) A vida real é confusa e, às vezes, uma solução simples para um problema não é viável. Nesse caso, utilize a solução mais simples possível que funcione.

```
Readability counts.
```

(**Legibilidade conta**) Mesmo quando seu código for complexo, procure deixá-lo legível. Quando trabalhar com um projeto que envolva códigos complexos, procure escrever comentários informativos para esse código.

```
There should be one-- and preferably only one --obvious way to do it.
```

(**Deve haver uma – e, de preferência, apenas uma – maneira óbvia de fazer algo**) Se for solicitado a dois programadores Python que resolvam o mesmo problema, eles deverão apresentar soluções razoavelmente compatíveis. Isso não quer dizer que não haja espaço para a criatividade em programação. Pelo contrário! No entanto, boa parte da programação consiste em usar abordagens pequenas e comuns para situações simples em um projeto maior e mais criativo. Os detalhes de funcionamento de seus programas devem fazer sentido para outros programadores Python.

```
Now is better than never.
```

(**Agora é melhor que nunca**) Você poderia passar o resto de sua vida conhecendo todas as complexidades de Python e da programação em geral, mas, nesse caso, jamais concluiria qualquer projeto. Não tente escrever um código perfeito; escreva um código que funcione e, então, decida se deve aperfeiçoá-lo nesse projeto ou passar para algo novo.

Ao prosseguir para o próximo capítulo e começar a explorar tópicos mais sofisticados, procure ter em mente essa filosofia de simplicidade e clareza. Programadores experientes respeitarão mais o seu código e ficarão felizes em oferecer feedback e colaborar com você em projetos interessantes.

FAÇA VOCÊ MESMO

2.11 – Zen de Python: Digite `import this` em uma sessão de terminal de Python e dê uma olhada nos princípios adicionais.

Resumo

Neste capítulo aprendemos a trabalhar com variáveis. Aprendemos a usar nomes descritivos para as variáveis e a resolver erros de nomes e de sintaxe quando surgirem. Vimos o que são strings e como exibi-las usando letras minúsculas, letras maiúsculas e iniciais maiúsculas. No início, utilizamos espaços em branco para organizar a saída e aprendemos a remover caracteres em branco desnecessários de diferentes partes de uma string. Começamos a trabalhar com inteiros e números de ponto flutuante, e lemos a respeito de alguns comportamentos inesperados com os quais devemos tomar cuidado quando trabalharmos com dados numéricos. Também aprendemos a escrever comentários explicativos para que você e outras pessoas leiam seu código com mais facilidade. Por fim, lemos a respeito da filosofia de manter seu código o mais simples possível, sempre que puder.

No Capítulo 3 aprenderemos a armazenar coleções de informações em variáveis chamadas *listas*. Veremos como percorrer uma lista, manipulando qualquer informação que ela tiver.

3
Introdução às listas

Neste capítulo e no próximo, veremos o que são listas e como começar a trabalhar com os elementos de uma lista. As listas permitem armazenar conjuntos de informações em um só lugar, independentemente de termos alguns itens ou milhões deles. As listas são um dos recursos mais eficazes de Python, prontamente acessíveis aos novos programadores, e elas agregam muitos conceitos importantes em programação.

O que é uma lista?

Uma *lista* é uma coleção de itens em uma ordem em particular. Podemos criar uma lista que inclua as letras do alfabeto, os dígitos de 0 a 9 ou os nomes de todas as pessoas de sua família. Você pode colocar qualquer informação que quiser em uma lista, e os itens de sua lista não precisam estar relacionados de nenhum modo em particular. Como uma lista geralmente contém mais de um elemento, é uma boa ideia deixar seu nome no plural, por exemplo, `letters`, `digits` ou `names`.

Em Python, colchetes (`[]`) indicam uma lista, e elementos individuais da lista são separados por vírgulas. Eis um exemplo simples de uma lista que contém alguns tipos de bicicleta: bicycles.py bicycles = ['trek', 'cannondale', 'redline', 'specialized']

```
print(bicycles) Se você pedir a Python que exiba uma lista, ela devolverá
a representação da lista, incluindo os colchetes: ['trek', 'cannondale',
'redline', 'specialized']
```

Como essa não é a saída que você quer que seus usuários vejam, vamos aprender a acessar os elementos individuais de uma lista.

Acessando elementos de uma lista

Listas são coleções ordenadas, portanto você pode acessar qualquer elemento de uma lista informando a posição – ou *índice* – do item desejado ao interpretador Python. Para acessar um elemento de uma lista, escreva o nome da lista seguido do índice do item entre colchetes.

Por exemplo, vamos extrair a primeira bicicleta da lista `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0]) A sintaxe para isso está em ❶. Quando solicitamos um
item simples de uma lista, Python devolve apenas esse elemento, sem
colchetes ou aspas: trek
```

Esse é o resultado que você quer que seus usuários vejam – uma saída limpa, formatada de modo elegante.

Também podemos usar os métodos de string do Capítulo 2 em qualquer elemento de uma lista. Por exemplo, podemos formatar o elemento `'trek'` de modo mais bonito usando o método `title()`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
  print(bicycles[0].title()) Esse exemplo gera a mesma saída do exemplo
  anterior, exceto pelo fato de 'Trek' começar com uma letra maiúscula.
```

A posição dos índices começa em 0, e não em 1

Python considera que o primeiro item de uma lista está na posição 0, e não na posição 1. Isso é válido para a maioria das linguagens de programação, e o motivo tem a ver com o modo como as operações em lista são implementadas em um nível mais baixo. Se estiver recebendo resultados inesperados, verifique se você não está cometendo um erro simples de deslocamento de um.

O segundo item de uma lista tem índice igual a 1. Usando esse sistema simples de contagem, podemos obter qualquer elemento que quisermos de uma lista subtraindo um de sua posição na lista. Por exemplo, para acessar o quarto item de uma lista, solicite o item no índice 3.

As instruções a seguir acessam as bicicletas nos índices `1` e `3`: `bicycles = ['trek', 'cannondale', 'redline', 'specialized']`

```
  print(bicycles[1]) print(bicycles[3]) Esse código devolve a segunda e a
  quarta bicicletas da lista: cannondale
  specialized
```

Python tem uma sintaxe especial para acessar o último elemento de uma lista. Ao solicitar o item no índice `-1`, Python sempre devolve o último item da lista: `bicycles = ['trek', 'cannondale', 'redline', 'specialized']`

print(bicycles[-1]) Esse código devolve o valor 'specialized'. Essa sintaxe é bem útil, pois, com frequência, você vai querer acessar os últimos itens de uma lista sem saber exatamente o tamanho dela. Essa convenção também se estende a outros valores negativos de índice. O índice -2 devolve o segundo item a partir do final da lista, o índice -3 devolve o terceiro item a partir do final, e assim sucessivamente.

Usando valores individuais de uma lista

Você pode usar valores individuais de uma lista, exatamente como faria com qualquer outra variável. Por exemplo, podemos usar concatenação para criar uma mensagem com base em um valor de uma lista.

Vamos tentar obter a primeira bicicleta da lista e compor uma mensagem usando esse valor.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = "My first bicycle was a " + bicycles[0].title() + "."
```

print(message) Em ❶, compomos uma frase usando o valor em bicycles[0] e a armazenamos na variável message. A saída é uma frase simples sobre a primeira bicicleta da lista: My first bicycle was a Trek.

FAÇA VOCÊ MESMO

Experimente criar estes programas pequenos para ter um pouco de experiência própria com listas em Python. Você pode criar uma nova pasta para os exercícios de cada capítulo a fim de mantê-los organizados.

3.1 – Nomes: Armazene os nomes de alguns de seus amigos em uma lista chamada **names**. Exiba o nome de cada pessoa acessando cada elemento da lista, um de cada vez.

3.2 – Saudações: Comece com a lista usada no Exercício 3.1, mas em vez de simplesmente exibir o nome de cada pessoa, apresente uma mensagem a elas. O texto de cada mensagem deve ser o mesmo, porém cada mensagem deve estar personalizada com o nome da pessoa.

3.3 – Sua própria lista: Pense em seu meio de transporte preferido, como motocicleta ou carro, e crie uma lista que armazene vários exemplos. Utilize sua lista para exibir uma série de frases sobre esses itens, como "Gostaria de ter uma moto Honda".

Alterando, acrescentando e removendo elementos

A maioria das listas que você criar será dinâmica, o que significa que você criará uma lista e então adicionará e removerá elementos dela à

medida que seu programa executar. Por exemplo, você poderia criar um jogo em que um jogador atire em alienígenas que caem do céu. Poderia armazenar o conjunto inicial de alienígenas em uma lista e então remover um item da lista sempre que um alienígena for atingido. Sempre que um novo alienígena aparecer na tela, adicione-o à lista. Sua lista de alienígenas diminuirá e aumentará de tamanho no decorrer do jogo.

Modificando elementos de uma lista

A sintaxe para modificar um elemento é semelhante à sintaxe para acessar um elemento de uma lista. Para alterar um elemento, use o nome da lista seguido do índice do elemento que você quer modificar e, então, forneça o novo valor que você quer que esse item tenha.

Por exemplo, vamos supor que temos uma lista de motocicletas, e que o primeiro item da lista seja `'honda'`. Como mudaríamos o valor desse primeiro item?

motorcycles.py ❶ *motorcycles = ['honda', 'yamaha', 'suzuki']*

```
  print(motorcycles)
❷ motorcycles[0] = 'ducati'
  print(motorcycles) O código em ❶ define a lista original, com 'honda'
  como o primeiro elemento. O código em ❷ altera o valor do primeiro item
  para 'ducati'. A saída mostra que o primeiro item realmente foi
  modificado e o restante da lista permaneceu igual: ['honda', 'yamaha',
  'suzuki']

  ['ducati', 'yamaha', 'suzuki']
```

Você pode mudar o valor de qualquer item de uma lista, e não apenas o primeiro.

Acrescentando elementos em uma lista

Você pode acrescentar um novo elemento em uma lista por diversos motivos. Por exemplo, talvez você queira que novos alienígenas apareçam no jogo, pode querer acrescentar novos dados em uma visualização ou adicionar novos usuários registrados em um site que você criou. Python oferece várias maneiras de acrescentar novos dados em listas existentes.

Concatenando elementos no final de uma lista

A maneira mais simples de acrescentar um novo elemento em uma lista é *concatenar* o item na lista. Quando concatenamos um item em uma lista, o novo elemento é acrescentado no final. Usando a mesma lista que tínhamos no exemplo anterior, adicionaremos o novo elemento `'ducati'` no final da lista: `motorcycles = ['honda', 'yamaha', 'suzuki']`

```
  print(motorcycles)
❶ motorcycles.append('ducati') print(motorcycles) O método append() em ❶
acrescenta 'ducati' no final da lista sem afetar qualquer outro elemento:
['honda', 'yamaha', 'suzuki']
  ['honda', 'yamaha', 'suzuki', 'ducati']
```

O método `append()` facilita criar listas dinamicamente. Por exemplo, podemos começar com uma lista vazia e então acrescentar itens à lista usando uma série de instruções `append()`. Usando uma lista vazia, vamos adicionar os elementos `'honda'`, `'yamaha'` e `'suzuki'` à lista: motorcycles = []

```
motorcycles.append('honda') motorcycles.append('yamaha')
motorcycles.append('suzuki')
print(motorcycles) A lista resultante tem exatamente o mesmo aspecto da
lista dos exemplos anteriores: ['honda', 'yamaha', 'suzuki']
```

Criar listas dessa maneira é bem comum, pois, com frequência, você não conhecerá os dados que seus usuários querem armazenar em um programa até que ele esteja executando. Para deixar que seus usuários tenham o controle, comece definindo uma lista vazia que armazenará os valores dos usuários. Em seguida, concatene cada novo valor fornecido à lista que você acabou de criar.

Inserindo elementos em uma lista

Você pode adicionar um novo elemento em qualquer posição de sua lista usando o método `insert()`. Faça isso especificando o índice do novo elemento e o valor do novo item.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
```

```
❶ motorcycles.insert(0, 'ducati') print(motorcycles) Nesse exemplo, o
código em ❶ insere o valor 'ducati' no início da lista. O método insert()
abre um espaço na posição 0 e armazena o valor 'ducati' nesse local. Essa
operação desloca todos os demais valores da lista uma posição à direita:
['ducati', 'honda', 'yamaha', 'suzuki']
```

Removendo elementos de uma lista

Com frequência, você vai querer remover um item ou um conjunto de itens de uma lista. Por exemplo, quando um jogador atinge um alienígena no céu com um tiro, é bem provável que você vá querer removê-lo da lista de alienígenas ativos. Se um usuário decidir cancelar a conta em sua aplicação web, você vai querer remover esse usuário da lista de usuários ativos. Você pode remover um item de acordo com sua posição na lista ou seu valor.

Removendo um item usando a instrução del

Se a posição do item que você quer remover de uma lista for conhecida, a instrução `del` poderá ser usada.

```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)
❶ del motorcycles[0]
  print(motorcycles) O código em ❶ usa del para remover o primeiro item,
  'honda', da lista de motocicletas: ['honda', 'yamaha', 'suzuki']

  ['yamaha', 'suzuki']
```

Você pode remover um item de qualquer posição em uma lista usando a instrução `del`, se souber qual é o seu índice. Por exemplo, eis o modo de remover o segundo item, `'yamaha'`, da lista: `motorcycles = ['honda', 'yamaha', 'suzuki']`

```
  print(motorcycles)
  del motorcycles[1]
  print(motorcycles) A segunda motocicleta é apagada da lista: ['honda',
  'yamaha', 'suzuki']

  ['honda', 'suzuki']
```

Nos dois exemplos não podemos mais acessar o valor que foi removido da lista após a instrução `del` ter sido usada.

Removendo um item com o método pop()

Às vezes, você vai querer usar o valor de um item depois de removê-lo de uma lista. Por exemplo, talvez você queira obter as posições *x* e *y* de um alienígena que acabou de ser atingido para que possa desenhar uma explosão nessa posição. Em uma aplicação web, você poderia remover um usuário de uma lista de membros ativos e então adicioná-lo a uma lista de membros inativos.

O método `pop()` remove o último item de uma lista, mas permite que você trabalhe com esse item depois da remoção. O termo *pop* deriva de pensar em uma lista como se fosse uma pilha de itens e remover um item (fazer um pop) do topo da pilha. Nessa analogia, o topo da pilha corresponde ao final da lista.

Vamos fazer um pop de uma motocicleta da lista de motocicletas: ❶ motorcycles = ['honda', 'yamaha', 'suzuki']

```
  print(motorcycles)
❷ popped_motorcycle = motorcycles.pop() ❸ print(motorcycles) ❹
print(popped_motorcycle) Começamos definindo e exibindo a lista motorcycles
em ❶. Em ❷ fazemos pop de um valor da lista e o armazenamos na variável
popped_motorcycle. Exibimos a lista em ❸ para mostrar que um valor foi
removido da lista. Então exibimos o valor removido em ❹ para provar que
ainda temos acesso ao valor removido.
```

A saída mostra que o valor `'suzuki'` foi removido do final da lista e agora está armazenado na variável `popped_motorcycle`: ['honda', 'yamaha', 'suzuki']

```
  ['honda', 'yamaha']
  suzuki
```

Como esse método `pop()` poderia ser útil? Suponha que as motocicletas da lista estejam armazenadas em ordem cronológica, de acordo com a época em que fomos seus proprietários. Se for esse o caso, podemos usar o método `pop()` para exibir uma afirmação sobre a última motocicleta que compramos: motorcycles = ['honda', 'yamaha', 'suzuki']

```
  last_owned = motorcycles.pop() print("The last motorcycle I owned was a "
  + last_owned.title() + ".") A saída é uma frase simples sobre a
  motocicleta mais recente que tivemos: The last motorcycle I owned was a
  Suzuki.
```

Removendo itens de qualquer posição em uma lista

Na verdade, você pode usar `pop()` para remover um item de qualquer

posição em uma lista se incluir o índice do item que você deseja remover entre parênteses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ first_owned = motorcycles.pop(0) ❷ print('The first motorcycle I owned
was a ' + first_owned.title() + '.') Começamos fazendo pop da primeira
motocicleta da lista em ❶ e, então, exibimos uma mensagem sobre essa
motocicleta em ❷. A saída é uma frase simples que descreve a primeira
motocicleta que eu tive: The first motorcycle I owned was a Honda.
```

Lembre-se de que, sempre que usar `pop()`, o item com o qual você trabalhar não estará mais armazenado na lista.

Se você não tiver certeza se deve usar a instrução `del` ou o método `pop()`, eis um modo fácil de decidir: quando quiser apagar um item de uma lista e esse item não vai ser usado de modo algum, utilize a instrução `del`; se quiser usar um item à medida que removê-lo, utilize o método `pop()`.

Removendo um item de acordo com o valor

Às vezes, você são saberá a posição do valor que quer remover de uma lista. Se conhecer apenas o valor do item que deseja remover, o método `remove()` poderá ser usado.

Por exemplo, vamos supor que queremos remover o valor `'ducati'` da lista de motocicletas.

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
❶ motorcycles.remove('ducati') print(motorcycles) O código em ❶ diz a
Python para descobrir em que lugar 'ducati' aparece na lista e remover esse
elemento: ['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

Também podemos usar o método `remove()` para trabalhar com um valor que está sendo removido de uma lista. Vamos remover o valor `'ducati'` e exibir um motivo para removê-lo da lista: ❶ `motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']`

```
print(motorcycles)
❷ too_expensive = 'ducati'
```

❸ motorcycles.remove(too_expensive) print(motorcycles) ❹ print("\nA " + too_expensive.title() + " is too expensive for me.") Após definir a lista em ❶, armazenamos o valor 'ducati' em uma variável chamada too_expensive ❷. Então utilizamos essa variável para dizer a Python qual valor deve ser removido da lista em ❸. Em ❹, o valor 'ducati' foi removido da lista, mas continua armazenado na variável too_expensive, permitindo exibir uma frase sobre o motivo pelo qual removemos 'ducati' da lista de motocicletas:

```
['honda', 'yamaha', 'suzuki', 'ducati']
  ['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

NOTA O método `remove()` apaga apenas a primeira ocorrência do valor que você especificar. Se houver a possibilidade de o valor aparecer mais de uma vez na lista, será necessário usar um laço para determinar se todas as ocorrências desse valor foram removidas. Aprenderemos a fazer isso no Capítulo 7.

FAÇA VOCÊ MESMO

Os exercícios a seguir são um pouco mais complexos que aqueles do Capítulo 2, porém darão a você a oportunidade de usar listas de todas as formas descritas.

3.4 – Lista de convidados: Se pudesse convidar alguém, vivo ou morto, para o jantar, quem você convidaria? Crie uma lista que inclua pelo menos três pessoas que você gostaria de convidar para jantar. Em seguida, utilize sua lista para exibir uma mensagem para cada pessoa, convidando-a para jantar.

3.5 – Alterando a lista de convidados: Você acabou de saber que um de seus convidados não poderá comparecer ao jantar, portanto será necessário enviar um novo conjunto de convites. Você deverá pensar em outra pessoa para convidar.

- Comece com seu programa do Exercício 3.4. Acrescente uma instrução `print` no final de seu programa, especificando o nome do convidado que não poderá comparecer.
- Modifique sua lista, substituindo o nome do convidado que não poderá comparecer pelo nome da nova pessoa que você está convidando.
- Exiba um segundo conjunto de mensagens com o convite, uma para cada pessoa que continua presente em sua lista.

3.6 – Mais convidados: Você acabou de encontrar uma mesa de jantar maior, portanto agora tem mais espaço disponível. Pense em mais três convidados para o jantar.

- Comece com seu programa do Exercício 3.4 ou do Exercício 3.5. Acrescente uma instrução `print` no final de seu programa informando às pessoas que você encontrou uma mesa de jantar maior.

- Utilize `insert()` para adicionar um novo convidado no início de sua lista.
- Utilize `insert()` para adicionar um novo convidado no meio de sua lista.
- Utilize `append()` para adicionar um novo convidado no final de sua lista.
- Exiba um novo conjunto de mensagens de convite, uma para cada pessoa que está em sua lista.

3.7 – Reduzindo a lista de convidados: Você acabou de descobrir que sua nova mesa de jantar não chegará a tempo para o jantar e tem espaço para somente dois convidados.

- Comece com seu programa do Exercício 3.6. Acrescente uma nova linha que mostre uma mensagem informando que você pode convidar apenas duas pessoas para o jantar.
- Utilize `pop()` para remover os convidados de sua lista, um de cada vez, até que apenas dois nomes permaneçam em sua lista. Sempre que remover um nome de sua lista, mostre uma mensagem a essa pessoa, permitindo que ela saiba que você sente muito por não poder convidá-la para o jantar.
- Apresente uma mensagem para cada uma das duas pessoas que continuam na lista, permitindo que elas saibam que ainda estão convidadas.
- Utilize `del` para remover os dois últimos nomes de sua lista, de modo que você tenha uma lista vazia. Mostre sua lista para garantir que você realmente tem uma lista vazia no final de seu programa.

Organizando uma lista

Com frequência, suas listas serão criadas em uma ordem imprevisível, pois nem sempre você poderá controlar a ordem em que seus usuários fornecem seus dados. Embora isso seja inevitável na maioria das circunstâncias, com frequência você vai querer apresentar suas informações em uma ordem em particular. Às vezes, você vai querer preservar a ordem original de sua lista, enquanto, em outras ocasiões, vai querer alterar essa ordem. Python oferece várias maneiras de organizar suas listas de acordo com a situação.

Ordenando uma lista de forma permanente com o método sort()

O método `sort()` de Python faz com que seja relativamente fácil ordenar uma lista. Suponha que temos uma lista de carros e queremos alterar a ordem da lista para armazenar os itens em ordem alfabética. Para simplificar essa tarefa, vamos supor que todos os valores da lista usam letras minúsculas.

cars.py cars = ['bmw', 'audi', 'toyota', 'subaru']

```
❶ cars.sort() print(cars)
```

O método `sort()` mostrado em ❶ altera a ordem da lista de forma permanente. Os carros agora estão em ordem alfabética e não podemos mais retornar à ordem original.

```
['audi', 'bmw', 'subaru', 'toyota']
```

Também podemos ordenar essa lista em ordem alfabética inversa, passando o argumento `reverse=True` ao método `sort()`. O exemplo a seguir ordena a lista de carros em ordem alfabética inversa: `cars = ['bmw', 'audi', 'toyota', 'subaru']`

```
cars.sort(reverse=True) print(cars)
```

Novamente, a ordem da lista foi permanentemente alterada: ['toyota', 'subaru', 'bmw', 'audi']

Ordenando uma lista temporariamente com a função sorted()

Para preservar a ordem original de uma lista, mas apresentá-la de forma ordenada, podemos usar a função `sorted()`. A função `sorted()` permite exibir sua lista em uma ordem em particular, mas não afeta a ordem propriamente dita da lista.

Vamos testar essa função na lista de carros.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']


❶ print("Here is the original list:") print(cars)


❷ print("\nHere is the sorted list:") print(sorted(cars))
❸ print("\nHere is the original list again:") print(cars)
```

Inicialmente, exibimos a lista em sua ordem original em ❶ e depois, em ordem alfabética em ❷. Depois que a lista é exibida na nova ordem, mostramos que a lista continua armazenada em sua ordem original em ❸.

```
Here is the original list: ['bmw', 'audi', 'toyota', 'subaru']


Here is the sorted list: ['audi', 'bmw', 'subaru', 'toyota']
```

```
❹ Here is the original list again: ['bmw', 'audi', 'toyota', 'subaru']
```

Observe que a lista preserva sua ordem original em ❹, depois que a função `sorted()` foi usada. Essa função também pode aceitar um argumento `reverse=True` se você quiser exibir uma lista em ordem alfabética inversa.

NOTA Ordenar uma lista em ordem alfabética é um pouco mais complicado quando todos os valores não utilizam letras minúsculas. Há várias maneiras de interpretar letras maiúsculas quando decidimos por uma sequência de ordenação, e especificar a ordem exata pode apresentar um nível de complexidade maior que aquele com que queremos lidar no momento. No entanto, a maior parte das abordagens à ordenação terá diretamente como base o que aprendemos nesta seção.

Exibindo uma lista em ordem inversa

Para inverter a ordem original de uma lista, podemos usar o método `reverse()`. Se armazenarmos originalmente a lista de carros em ordem cronológica, de acordo com a época em que fomos seus proprietários, poderemos facilmente reorganizar a lista em ordem cronológica inversa:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
  print(cars)


  cars.reverse() print(cars)
```

Observe que `reverse()` não reorganiza em ordem alfabética inversa; ele simplesmente inverte a ordem da lista: ['bmw', 'audi', 'toyota', 'subaru']

```
  ['subaru', 'toyota', 'audi', 'bmw']
```

O método `reverse()` muda a ordem de uma lista de forma permanente, mas podemos restaurar a ordem original a qualquer momento aplicando `reverse()` à mesma lista uma segunda vez.

Descobrindo o tamanho de uma lista

Podemos rapidamente descobrir o tamanho de uma lista usando a função `len()`. A lista no exemplo a seguir tem quatro itens, portanto seu tamanho é 4: >>> **`cars = ['bmw', 'audi', 'toyota', 'subaru']`**

```
  >>> len(cars) 4
```

Você achará `len()` útil quando precisar identificar o número de alienígenas que ainda precisam ser atingidos em um jogo, determinar a quantidade de dados que você precisa administrar em uma visualização

ou descobrir o número de usuários registrados em um site, entre outras tarefas.

NOTA Python conta os itens de uma lista começando em um, portanto você não deverá se deparar com nenhum erro de deslocamento de um ao determinar o tamanho de uma lista.

FAÇA VOCÊ MESMO

3.8 – Conhecendo o mundo: Pense em pelo menos cinco lugares do mundo que você gostaria de visitar.

- Armazene as localidades em uma lista. Certifique-se de que a lista não esteja em ordem alfabética.
- Exiba sua lista na ordem original. Não se preocupe em exibir a lista de forma elegante; basta exibi-la como uma lista Python pura.
- Utilize `sorted()` para exibir sua lista em ordem alfabética, sem modificar a lista propriamente dita.
- Mostre que sua lista manteve sua ordem original exibindo-a.
- Utilize `sorted()` para exibir sua lista em ordem alfabética inversa sem alterar a ordem da lista original.
- Mostre que sua lista manteve sua ordem original exibindo-a novamente.
- Utilize `reverse()` para mudar a ordem de sua lista. Exiba a lista para mostrar que sua ordem mudou.
- Utilize `reverse()` para mudar a ordem de sua lista novamente. Exiba a lista para mostrar que ela voltou à sua ordem original.
- Utilize `sort()` para mudar sua lista de modo que ela seja armazenada em ordem alfabética. Exiba a lista para mostrar que sua ordem mudou.
- Utilize `sort()` para mudar sua lista de modo que ela seja armazenada em ordem alfabética inversa. Exiba a lista para mostrar que sua ordem mudou.

3.9 – Convidados para o jantar: Trabalhando com um dos programas dos Exercícios de 3.4 a 3.7 (páginas 80 e 81), use `len()` para exibir uma mensagem informando o número de pessoas que você está convidando para o jantar.

3.10 – Todas as funções: Pensa em algo que você poderia armazenar em uma lista. Por exemplo, você poderia criar uma lista de montanhas, rios, países, cidades, idiomas ou qualquer outro item que quiser. Escreva um programa que crie uma lista contendo esses itens e então utilize cada função apresentada neste capítulo pelo menos uma vez.

Evitando erros de índice quando trabalhar com listas

Um tipo de erro é comum quando trabalhamos com listas pela primeira

vez. Suponha que temos uma lista com três itens e você solicite o quarto item: motorcycles = ['honda', 'yamaha', 'suzuki']

```
print(motorcycles[3]) Esse exemplo resulta em um erro de índice:
Traceback (most recent call last): File "motorcycles.py", line 3, in
<module> print(motorcycles[3]) IndexError: list index out of range Python
tenta fornecer o item no índice 3. Porém, quando pesquisa a lista, nenhum
item de motorcycles tem índice igual a 3. Por causa da natureza deslocada
de um na indexação de listas, esse erro é característico. As pessoas
acham que o terceiro item é o item de número 3, pois começam a contar a
partir de 1. Contudo, em Python, o terceiro item é o de número 2, pois a
indexação começa em 0.
```

Um erro de índice quer dizer que Python não é capaz de determinar o índice solicitado. Se um erro de índice ocorrer em seu programa, tente ajustar o índice que você está solicitando em um. Então execute o programa novamente para ver se os resultados estão corretos.

Tenha em mente que, sempre que quiser acessar o último item de uma lista, você deve usar o índice -1. Isso sempre funcionará, mesmo que sua lista tenha mudado de tamanho desde a última vez que você a acessou:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1]) O índice -1 sempre devolve o último item de uma
lista - nesse caso, é o valor 'suzuki': 'suzuki'
```

A única ocasião em que essa abordagem causará um erro é quando solicitamos o último item de uma lista vazia: motorcycles = []

```
print(motorcycles[-1]) Não há nenhum item em motorcycles, portanto Python
devolve outro erro de índice: Traceback (most recent call last): File
"motorcyles.py", line 3, in <module> print(motorcycles[-1]) IndexError:
list index out of range NOTA Se um erro de índice ocorrer e você não
consegue descobrir como resolvê-lo, experimente exibir sua lista ou
simplesmente mostrar o tamanho dela. Sua lista poderá estar bem diferente
do que você imaginou, em especial se ela foi tratada dinamicamente pelo
seu programa. Ver a lista propriamente dita ou o número exato de itens
que ela contém pode ajudar a entender esses erros de lógica.
```

FAÇA VOCÊ MESMO

3.11 – Erro proposital: Se você ainda não recebeu um erro de índice em um de seus programas, tente fazer um erro desse tipo acontecer. Altere um índice em um de seus programas de modo a gerar um erro de índice. Não se esqueça de corrigir o erro antes de fechar o programa.

Resumo

Neste capítulo conhecemos as listas e vimos como trabalhar com os itens individuais de uma lista. Aprendemos a definir uma lista e a adicionar e remover elementos. Vimos como ordenar as listas de modo permanente e temporário para fins de exibição. Também vimos como descobrir o tamanho de uma lista e aprendemos a evitar erros de índice quando trabalhamos com listas.

No Capítulo 4 veremos como trabalhar com itens de uma lista de maneira mais eficiente. Ao percorrer todos os itens em um laço usando apenas algumas linhas de código, poderemos trabalhar de modo eficiente, mesmo quando a lista contiver milhares ou milhões de itens.

4
Trabalhando com listas

No Capítulo 3 aprendemos a criar uma lista simples e a trabalhar com os elementos individuais de uma lista. Neste capítulo veremos como percorrer uma lista inteira com um *laço* usando apenas algumas linhas de código, independentemente do tamanho da lista. Percorrer listas com laços permite executar a mesma ação – ou conjunto de ações – em todos os itens de uma lista. Como resultado, você poderá trabalhar de modo eficiente com listas de qualquer tamanho, incluindo aquelas com milhares ou até mesmo milhões de itens.

Percorrendo uma lista inteira com um laço

Com frequência, você vai querer percorrer todas as entradas de uma lista, executando a mesma tarefa em cada item. Por exemplo, em um jogo, você pode mover todos os elementos da tela de acordo com a mesma distância; em uma lista de números, talvez você queira executar a mesma operação estatística em todos os elementos. Quem sabe você queira exibir cada um dos títulos de uma lista de artigos em um site. Quando quiser executar a mesma ação em todos os itens de uma lista, você pode usar o laço `for` de Python.

Vamos supor que temos uma lista de nomes de mágicos e queremos exibir todos os nomes da lista. Poderíamos fazer isso recuperando cada nome da lista individualmente, mas essa abordagem poderia causar vários problemas. Para começar, seria repetitivo fazer isso com uma lista longa de nomes. Além disso, teríamos que alterar o nosso código sempre que o tamanho da lista mudasse. Um laço `for` evita esses dois problemas ao permitir que Python administre essas questões internamente.

Vamos usar um laço `for` para exibir cada um dos nomes de uma lista de mágicos: magicians.py ❶ magicians = ['alice', 'david', 'carolina']

```
❷ for magician in magicians: ❸ print(magician) Começamos definindo uma
lista em ❶, exatamente como fizemos no Capítulo 3. Em ❷ definimos um laço
for. Essa linha diz a Python para extrair um nome da lista magicians e
armazená-lo na variável magician. Em ❸ dizemos a Python para exibir o nome
que acabou de ser armazenado em magician. O interpretador então repete as
linhas ❷ e ❸, uma vez para cada nome da lista. Ler esse código como "para
todo mágico na lista de mágicos, exiba o nome do mágico" pode ajudar. A
saída é uma exibição simples de cada nome da lista: alice
  david
  carolina
```

Observando os laços com mais detalhes

O conceito de laços é importante porque é uma das maneiras mais comuns para um computador automatizar tarefas repetitivas. Por exemplo, em um laço simples como o que usamos em *magicians.py*, Python inicialmente lê a primeira linha do laço: for magician in magicians: Essa linha diz a Python para extrair o primeiro valor da lista `magicians` e armazená-lo na variável `magician`. O primeiro valor é `'alice'`. O interpretador então lê a próxima linha: print(magician) Python exibe o valor atual de `magician`, que ainda é `'alice'`. Como a lista contém mais valores, o interpretador retorna à primeira linha do laço: for magician in magicians: Python recupera o próximo nome da lista, que é `'david'`, e armazena esse valor em `magician`. Então ele executa a linha: print(magician) Python exibe o valor atual de `magician`, que agora é `'david'`, novamente. O interpretador repete todo o laço mais uma vez com o último valor da lista, que é `'carolina'`. Como não há mais valores na lista, Python passa para a próxima linha do programa. Nesse caso, não há mais nada depois do laço `for`, portanto o programa simplesmente termina.

Quando usar laços pela primeira vez, tenha em mente que o conjunto de passos será repetido, uma vez para cada item da lista, não importa quantos itens haja na lista. Se você tiver um milhão de itens em sua lista, Python repetirá esses passos um milhão de vezes – e geralmente o fará bem rápido.

Tenha em mente também que quando escrever seus próprios laços `for`, você poderá escolher qualquer nome que quiser para a variável temporária que armazena cada valor da lista. No entanto, é conveniente escolher um nome significativo, que represente um único item da lista.

Por exemplo, eis uma boa maneira de iniciar um laço `for` para uma lista de gatos, uma lista de cachorros e uma lista genérica de itens: for cat in cats: for dog in dogs: for item in list_of_items: Essas convenções de nomenclatura podem ajudar você a acompanhar a ação executada em cada item em um laço `for`. O uso de nomes no singular e no plural pode ajudar a identificar se a seção de código atua em um único elemento da lista ou em toda a lista.

Executando mais tarefas em um laço for

Você pode fazer praticamente de tudo com cada item em um laço `for`. Vamos expandir o exemplo anterior exibindo uma mensagem a cada mágico, informando-lhes que realizaram um ótimo truque: `magicians = ['alice', 'david', 'carolina']`

```
for magician in magicians: ❶ print(magician.title() + ", that was a
great trick!") A única diferença nesse código está em ❶, em que compomos
uma mensagem para cada mágico, começando com o nome desse mágico. Na
primeira vez em que passamos pelo laço, o valor do mágico é 'alice',
portanto Python inicia a primeira mensagem com o nome 'Alice'. Na segunda
vez que compormos a mensagem, ela começará com 'David' e na terceira vez,
a mensagem começará com 'Carolina'.
```

A saída mostra uma mensagem personalizada para cada mágico da lista: Alice, that was a great trick!

```
David, that was a great trick!
Carolina, that was a great trick!
```

Também podemos escrever tantas linhas de código quantas quisermos no laço `for`. Considera-se que toda linha indentada após a linha `for magician in magicians` está *dentro do laço*, e cada linha indentada é executada uma vez para cada valor da lista. Assim, você pode executar o volume de trabalho que quiser com cada valor da lista.

Vamos acrescentar uma segunda linha em nossa mensagem, informando a cada mágico que estamos ansiosos para ver o seu próximo truque: `magicians = ['alice', 'david', 'carolina']`

```
for magician in magicians: print(magician.title() + ", that was a great
trick!") ❶ print("I can't wait to see your next trick, " +
magician.title() + ".\n") Como indentamos as duas instruções print, cada
linha será executada uma vez para cada mágico da lista. A quebra de linha
("\n") na segunda instrução print ❶ insere uma linha em branco após cada
passagem pelo laço. Com isso, criamos um conjunto de mensagens agrupadas
de forma organizada para cada pessoa na lista: Alice, that was a great
trick!
I can't wait to see your next trick, Alice.


David, that was a great trick!
I can't wait to see your next trick, David.


Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Podemos usar quantas linhas quisermos em nossos laços `for`. Na prática, muitas vezes você achará útil efetuar várias operações com cada item de uma lista quando usar um laço `for`.

Fazendo algo após um laço for

O que acontece quando um laço `for` acaba de executar? Geralmente você vai querer fazer uma síntese de um bloco de saída ou passar para outra atividade que seu programa deva executar.

Qualquer linha de código após o laço `for` que não estiver indentada será executada uma vez, sem repetição. Vamos escrever um agradecimento ao grupo de mágicos como um todo, agradecendo-lhes por apresentar um show excelente. Para exibir essa mensagem ao grupo após todas as mensagens individuais terem sido apresentadas, colocamos a mensagem de agradecimento depois do laço `for`, sem indentação: `magicians = ['alice', 'david', 'carolina']`

```
for magician in magicians: print(magician.title() + ", that was a great
trick!") print("I can't wait to see your next trick, " + magician.title()
+ ".\n")
```

```
❶ print("Thank you, everyone. That was a great magic show!") As duas
primeiras instruções print são repetidas uma vez para cada mágico da lista,
como vimos antes. No entanto, como a linha em ❶ não está indentada, ela
será exibida apenas uma vez: Alice, that was a great trick!

I can't wait to see your next trick, Alice.
```

```
David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

Quando processar dados com um laço `for`, você verá que essa é uma boa maneira de sintetizar uma operação realizada em todo um conjunto de dados. Por exemplo, um laço `for` pode ser usado para inicializar um jogo percorrendo uma lista de personagens e exibindo cada um deles na tela. Você pode então escrever um bloco não indentado após esse laço, que exiba um botão Play Now (Jogue agora) depois que todos os personagens tiverem sido desenhados na tela.

Evitando erros de indentação

Python usa indentação para determinar se uma linha de código está conectada à linha antes dela. Nos exemplos anteriores, as linhas que exibiam mensagens aos mágicos individuais faziam parte do laço `for` porque estavam indentadas. O uso de indentação por Python deixa o código bem fácil de ler. Basicamente, Python usa espaços em branco para forçar você a escrever um código formatado de modo organizado, com uma estrutura visual clara. Em programas Python mais longos, você perceberá que há blocos de código indentados em alguns níveis diferentes. Esses níveis de indentação ajudam a ter uma noção geral da organização do programa como um todo.

Quando começar a escrever código que dependa de uma indentação apropriada, você deverá tomar cuidado com alguns *erros comuns de indentação*. Por exemplo, às vezes, as pessoas indentam blocos de código que não precisam estar indentados ou se esquecem de indentar blocos que deveriam estar indentados. Ver exemplos desses erros agora ajudará a evitá-los no futuro e a corrigi-los quando aparecerem em seus próprios programas.

Vamos analisar alguns dos erros mais comuns de indentação.

Esquecendo-se de indentar

Sempre indente a linha após a instrução `for` em um laço. Se você se esquecer, Python o avisará: magicians.py magicians = ['alice', 'david', 'carolina']

```
for magician in magicians: ❶ print(magician) A instrução print em ❶
deveria estar indentada, mas não está. Quando Python espera um bloco
indentado e não encontra um, ele mostra a linha em que o problema
ocorreu.

  File "magicians.py", line 3
  print(magician) ^
IndentationError: expected an indented block Geralmente podemos resolver
esse tipo de erro indentando a linha ou as linhas logo depois da
instrução for.
```

Esquecendo-se de indentar linhas adicionais

Às vezes, seu laço executará sem erros, mas não produzirá o resultado esperado. Isso pode acontecer quando você tenta realizar várias tarefas em um laço e se esquece de indentar algumas de suas linhas.

Por exemplo, eis o que acontece quando nos esquecemos de indentar a segunda linha do laço que diz a cada mágico que estamos ansiosos para ver o seu próximo truque: magicians = ['alice', 'david', 'carolina']

```
for magician in magicians: print(magician.title() + ", that was a great
trick!") ❶ print("I can't wait to see your next trick, " +
magician.title() + ".\n") A instrução print em ❶ deveria estar
indentada, mas como Python encontra pelo menos uma linha indentada após a
instrução for, um erro não é informado. Como resultado, a primeira
instrução print é executada uma vez para cada nome da lista, pois está
indentada. A segunda instrução print não está indentada, portanto é
executada somente uma vez, depois que o laço terminar de executar. Como o
valor final de magician é 'carolina', ela é a única que recebe a mensagem
de “ansiosos pelo próximo truque”: Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Esse é um *erro de lógica*. É um código Python com sintaxe válida, mas ele não gera o resultado desejado, pois há um problema em sua lógica. Se você espera ver uma determinada ação ser repetida uma vez para cada item de uma lista, mas ela é executada apenas uma vez, verifique se não é preciso simplesmente indentar uma linha ou um grupo de linhas.

Indentando desnecessariamente

Se você, acidentalmente, indentar uma linha que não precisa ser indetada, Python o informará a respeito da indentação inesperada: hello_world.py message = "Hello Python world!"

❶ print(message) Não precisamos indentar a instrução print em ❶, pois ela não *pertence* à linha antes dela; assim, Python informa esse erro: File "hello_world.py", line 2

```
    print(message) ^
  IndentationError: unexpected indent Você pode evitar erros inesperados de
  indentação ao indentar apenas quando houver um motivo específico para
  isso. Nos programas que estamos escrevendo no momento, as únicas linhas
  que precisam ser indentadas são as ações que queremos repetir para cada
  item em um laço for.
```

Indentando desnecessariamente após o laço

Se você acidentalmente indentar um código que deva executar após um laço ter sido concluído, esse código será repetido uma vez para cada item da lista. Às vezes, isso faz Python informar um erro, mas, geralmente, você terá um erro de lógica simples.

Por exemplo, vamos ver o que acontece quando indentamos por acidente a linha que agradece aos mágicos como um grupo por apresentarem um bom show: magicians = ['alice', 'david', 'carolina']

```
  for magician in magicians: print(magician.title() + ", that was a great
  trick!") print("I can't wait to see your next trick, " + magician.title()
  + ".\n")
```

❶ print("Thank you everyone, that was a great magic show!") Pelo fato de a linha em ❶ estar indentada, ela é exibida uma vez para cada pessoa da lista, como podemos ver em ❷: Alice, that was a great trick!

```
  I can't wait to see your next trick, Alice.


❷ Thank you everyone, that was a great magic show!
  David, that was a great trick!
  I can't wait to see your next trick, David.


❷ Thank you everyone, that was a great magic show!
  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.
```

```
❷ Thank you everyone, that was a great magic show!
```

Há outro erro de lógica, semelhante àquele da seção "Esquecendo-se de indentar linhas adicionais". Como Python não sabe o que você está querendo fazer com seu código, ele executará todo código que estiver escrito com uma sintaxe válida. Se uma ação for repetida muitas vezes quando deveria ser executada apenas uma vez, verifique se você não precisa simplesmente deixar de indentar o código dessa ação.

Esquecendo os dois-pontos

Os dois-pontos no final de uma instrução `for` diz a Python para interpretar a próxima linha como o início de um laço.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians print(magician) Se você se esquecer
acidentalmente de colocar os dois-pontos, como em ❶, você terá um erro de
sintaxe, pois Python não sabe o que você está tentando fazer. Embora esse
seja um erro fácil de corrigir, nem sempre é um erro fácil de identificar.
Você ficaria surpreso com a quantidade de tempo gasto pelos programadores à
procura de erros de um único caractere como esse. Erros desse tipo são
difíceis de encontrar, pois, com frequência, vemos somente aquilo que
esperamos ver.
```

FAÇA VOCÊ MESMO

4.1 – Pizzas: Pense em pelo menos três tipos de pizzas favoritas. Armazene os nomes dessas pizzas e, então, utilize um laço `for` para exibir o nome de cada pizza.

- Modifique seu laço `for` para mostrar uma frase usando o nome da pizza em vez de exibir apenas o nome dela. Para cada pizza, você deve ter uma linha na saída contendo uma frase simples como *Gosto de pizza de pepperoni*.
- Acrescente uma linha no final de seu programa, fora do laço `for`, que informe quanto você gosta de pizza. A saída deve ser constituída de três ou mais linhas sobre os tipos de pizza que você gosta e de uma frase adicional, por exemplo, *Eu realmente adoro pizza!*

4.2 – Animais: Pense em pelo menos três animais diferentes que tenham uma característica em comum. Armazene os nomes desses animais em uma lista e, então, utilize um laço `for` para exibir o nome de cada animal.

- Modifique seu programa para exibir uma frase sobre cada animal, por exemplo, *Um cachorro seria um ótimo animal de estimação.*
- Acrescente uma linha no final de seu programa informando o que esses animais têm em comum. Você poderia exibir uma frase como *Qualquer um desses animais seria um ótimo animal de estimação!*

Criando listas numéricas

Há muitos motivos para armazenar um conjunto de números. Por exemplo, você precisará manter um controle das posições de cada personagem em um jogo, e talvez queira manter um registro das pontuações mais altas de um jogador também. Em visualizações de dados, quase sempre você trabalhará com conjuntos de números, como temperaturas, distâncias, tamanhos de população ou valores de latitudes e longitudes, entre outros tipos de conjuntos numéricos.

As listas são ideais para armazenar conjuntos de números, e Python oferece várias ferramentas para ajudar você a trabalhar com listas de números de forma eficiente. Depois que souber usar efetivamente essas ferramentas, seu código funcionará bem, mesmo quando suas listas tiverem milhões de itens.

Usando a função range()

A função `range()` de Python facilita gerar uma série de números. Por exemplo, podemos usar a função `range()` para exibir uma sequência de números, assim: numbers.py for value in range(1,5): print(value) Embora esse código dê a impressão de que deveria exibir os números de 1 a 5, ele não exibe o número 5: 1

```
2
3
4
```

Nesse exemplo, `range()` exibe apenas os números de 1 a 4. Esse é outro resultado do comportamento deslocado de um que veremos com frequência nas linguagens de programação. A função `range()` faz Python começar a contar no primeiro valor que você lhe fornecer e parar quando atingir o segundo valor especificado. Como ele para nesse segundo valor, a saída não conterá o valor final, que seria 5, nesse caso.

Para exibir os números de 1 a 5, você deve usar `range(1,6)`: for value in range(1,6): `print(value)` Dessa vez, a saída começa em 1 e termina em 5: 1

```
2
3
4
5
```

Se sua saída for diferente do esperado ao usar `range()`, experimente ajustar seu valor final em 1.

Usando range() para criar uma lista de números

Se quiser criar uma lista de números, você pode converter os resultados de `range()` diretamente em uma lista usando a função `list()`. Quando colocamos `list()` em torno de uma chamada à função `range()`, a saída será uma lista de números.

No exemplo da seção anterior, simplesmente exibimos uma série de números. Podemos usar `list()` para converter esse mesmo conjunto de números em uma lista: numbers = list(range(1,6)) print(numbers) E o resultado será este: [1, 2, 3, 4, 5]

Também podemos usar a função `range()` para dizer a Python que ignore alguns números em um dado intervalo. Por exemplo, eis o modo de listar os números pares entre 1 e 10: even_numbers.py even_numbers = list(range(2,11,2)) print(even_numbers) Nesse exemplo, a função `range()` começa com o valor 2 e então soma 2 a esse valor. O valor 2 é somado repetidamente até o valor final, que é 11, ser alcançado ou ultrapassado, e o resultado a seguir é gerado: [2, 4, 6, 8, 10]

Podemos criar praticamente qualquer conjunto de números que quisermos com a função `range()`. Por exemplo, considere como criaríamos uma lista dos dez primeiros quadrados perfeitos (isto é, o quadrado de cada inteiro de 1 a 10). Em Python, dois asteriscos (`**`) representam exponenciais. Eis o modo como podemos colocar os dez primeiros quadrados perfeitos em uma lista: squares.py ❶ squares = []
❷ for value in range(1,11): ❸ square = value**2
❹ squares.append(square)
❺ print(squares) Começamos com uma lista vazia chamada squares em ❶. Em ❷, dizemos a Python para percorrer cada valor de 1 a 10 usando a função range(). No laço, o valor atual é elevado ao quadrado e armazenado na variável square em ❸. Em ❹, cada novo valor de square é concatenado à lista squares. Por fim, quando o laço acaba de executar, a lista de quadrados é exibida em ❺: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Para escrever esse código de modo mais conciso, omita a variável temporária `square` e concatene cada novo valor diretamente na lista:

```
squares = []
  for value in range(1,11): ❶ squares.append(value**2)
```

```
print(squares) O código em ❶ faz a mesma tarefa executada pelas linhas
❸ e ❹ em squares.py. Cada valor do laço é elevado ao quadrado e, então,
é imediatamente concatenado à lista de quadrados.
```

Você pode usar qualquer uma dessas duas abordagens quando criar listas mais complexas. Às vezes, usar uma variável temporária deixa o código mais legível; em outras ocasiões, deixa o código desnecessariamente longo. Concentre-se primeiro em escrever um código que você entenda claramente e faça o que você quer que ele faça. Em seguida, procure abordagens mais eficientes à medida que revisar seu código.

Estatísticas simples com uma lista de números

Algumas funções Python são específicas para listas de números. Por exemplo, podemos encontrar facilmente o valor mínimo, o valor máximo e a soma de uma lista de números: >>> **digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]**

```
>>> min(digits) 0
>>> max(digits) 9
>>> sum(digits) 45
```

NOTA Os exemplos desta seção utilizam listas pequenas de números para que caibam facilmente na página. Esses exemplos também funcionarão bem se sua lista contiver um milhão de números ou mais.

List comprehensions

A abordagem descrita antes para gerar a lista `squares` usou três ou quatro linhas de código. Uma *list comprehension* (abrangência de lista) permite gerar essa mesma lista com apenas uma linha de código. Uma list comprehension combina o laço `for` e a criação de novos elementos em uma linha, e concatena cada novo elemento automaticamente. As list comprehensions nem sempre são apresentadas aos iniciantes, mas eu as incluí aqui porque é bem provável que você as veja assim que começar a analisar códigos de outras pessoas.

O exemplo a seguir cria a mesma lista de quadrados perfeitos que vimos antes, porém utiliza uma list comprehension: squares.py squares = [value**2 for value in range(1,11)]

```
print(squares) Para usar essa sintaxe, comece com um nome descritivo para
a lista, por exemplo, squares. Em seguida, insira um colchete de abertura
e defina a expressão para os valores que você quer armazenar na nova
lista. Nesse exemplo, a expressão é value**2, que eleva o valor ao
quadrado. Então escreva um laço for para gerar os números que você quer
fornecer à expressão e insira um colchete de fechamento. O laço for nesse
exemplo é for value in range(1,11), que fornece os valores de 1 a 10 à
expressão value**2. Observe que não usamos dois-pontos no final da
instrução for.
```

O resultado é a mesma lista de valores ao quadrado que vimos antes: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Escrever suas próprias list comprehensions exige um pouco de prática, mas você verá que vale a pena conhecê-las depois que se sentir à vontade para criar listas comuns. Quando escrever três ou quatro linhas de código para gerar listas e isso começar a parecer repetitivo, considere escrever suas próprias list comprehensions.

FAÇA VOCÊ MESMO

4.3 – Contando até vinte: Use um laço `for` para exibir os números de 1 a 20, incluindo-os.

4.4 – Um milhão: Crie uma lista de números de um a um milhão e, então, use um laço `for` para exibir os números. (Se a saída estiver demorando demais, interrompa pressionando CTRL-C ou feche a janela de saída.) **4.5 – Somando um milhão:** Crie uma lista de números de um a um milhão e, em seguida, use `min()` e `max()` para garantir que sua lista realmente começa em um e termina em um milhão. Além disso, utilize a função `sum()` para ver a rapidez com que Python é capaz de somar um milhão de números.

4.6 – Números ímpares: Use o terceiro argumento da função `range()` para criar uma lista de números ímpares de 1 a 20. Utilize um laço `for` para exibir todos os números.

4.7– Três: Crie uma lista de múltiplos de 3, de 3 a 30. Use um laço `for` para exibir os números de sua lista.

4.8 – Cubos: Um número elevado à terceira potência é chamado de *cubo*. Por exemplo, o cubo de 2 é escrito como `2**3` em Python. Crie uma lista dos dez primeiros cubos (isto é, o cubo de cada inteiro de 1 a 10), e utilize um laço `for` para exibir o valor de cada cubo.

4.9 – Comprehension de cubos: Use uma list comprehension para gerar uma lista dos dez primeiros cubos.

Trabalhando com parte de uma lista

No Capítulo 3 aprendemos a acessar elementos únicos de uma lista e, neste capítulo, aprendemos a trabalhar com todos os elementos de uma lista. Também podemos trabalhar com um grupo específico de itens de uma lista, que Python chama de *fatia*.

Fatiando uma lista

Para criar uma fatia, especifique o índice do primeiro e do último elemento com os quais você quer trabalhar. Como ocorre na função `range()`, Python para em um item antes do segundo índice que você especificar. Para exibir os três primeiros elementos de uma lista, solicite os índices de `0` a `3`; os elementos `0`, `1` e `2` serão devolvidos.

O exemplo a seguir envolve uma lista de jogadores de um time: players.py players = ['charles', 'martina', 'michael', 'florence', 'eli']

```
❶ print(players[0:3]) O código em ❶ exibe uma fatia dessa lista, que
inclui apenas os três primeiros jogadores. A saída mantém a estrutura de
lista e inclui os três primeiros jogadores: ['charles', 'martina',
'michael']
```

Você pode gerar qualquer subconjunto de uma lista. Por exemplo, se quiser o segundo, o terceiro e o quarto itens de uma lista, comece a fatia no índice `1` e termine no índice `4`: `players = ['charles', 'martina', 'michael', 'florence', 'eli']`

```
print(players[1:4]) Dessa vez, a fatia começa com 'martina' e termina com
'florence': ['martina', 'michael', 'florence']
```

Se o primeiro índice de uma fatia for omitido, Python começará sua fatia automaticamente no início da lista: `players = ['charles', 'martina', 'michael', 'florence', 'eli']`

```
print(players[:4]) Sem um índice de início, Python usa o início da lista:
['charles', 'martina', 'michael', 'florence']
```

Uma sintaxe semelhante funcionará se você quiser uma fatia que inclua o final de uma lista. Por exemplo, se quiser todos os itens do terceiro até o último item, podemos começar com o índice `2` e omitir o segundo índice: `players = ['charles', 'martina', 'michael', 'florence', 'eli']`

```
print(players[2:]) Python devolve todos os itens, do terceiro item até o
final da lista: ['michael', 'florence', 'eli']
```

Essa sintaxe permite apresentar todos os elementos a partir de

qualquer ponto de sua lista até o final, independentemente do tamanho da lista. Lembre-se de que um índice negativo devolve um elemento a uma determinada distância do final de uma lista; assim, podemos exibir qualquer fatia a partir do final de uma lista. Por exemplo, se quisermos apresentar os três últimos jogadores da lista, podemos usar a fatia `players[-3:]`: `players = ['charles', 'martina', 'michael', 'florence', 'eli']`

`print(players[-3:])` Esse código exibe os nomes dos três últimos jogadores e continuaria a funcionar à medida que a lista de jogadores mudar de tamanho.

Percorrendo uma fatia com um laço

Você pode usar uma fatia em um laço `for` se quiser percorrer um subconjunto de elementos de uma lista. No próximo exemplo, percorreremos os três primeiros jogadores e exibiremos seus nomes como parte de uma lista simples: players = ['charles', 'martina', 'michael', 'florence', 'eli']

```
print("Here are the first three players on my team:") ❶ for player in
players[:3]: print(player.title()) Em vez de percorrer a lista inteira de
jogadores em ❶, Python percorre somente os três primeiros nomes: Here
are the first three players on my team: Charles
Martina
Michael
```

As fatias são muito úteis em várias situações. Por exemplo, quando criar um jogo, você poderia adicionar a pontuação final de um jogador em uma lista sempre que esse jogador acabar de jogar. Seria possível então obter as três pontuações mais altas de um jogador ordenando a lista em ordem decrescente e obtendo uma lista que inclua apenas as três primeiras pontuações. Ao trabalhar com dados, você pode usar fatias para processar seus dados em porções de tamanho específico. Quando criar uma aplicação web, fatias poderiam ser usadas para exibir informações em uma série de páginas, com uma quantidade apropriada de informações em cada página.

Copiando uma lista

Com frequência, você vai querer começar com uma lista existente e criar

uma lista totalmente nova com base na primeira. Vamos explorar o modo de copiar uma lista e analisar uma situação em que copiar uma lista é útil.

Para copiar uma lista, podemos criar uma fatia que inclua a lista original inteira omitindo o primeiro e o segundo índices (`[:]`). Isso diz a Python para criar uma lista que começa no primeiro item e termina no último, gerando uma cópia da lista toda.

Por exemplo, suponha que temos uma lista de nossos alimentos prediletos e queremos criar uma lista separada de comidas que um amigo gosta. Esse amigo gosta de tudo que está em nossa lista até agora, portanto podemos criar sua lista copiando a nossa: foods.py ❶
my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

print("My favorite foods are:") print(my_foods)
print("\nMy friend's favorite foods are:") print(friend_foods) Em ❶ criamos uma lista de alimentos de que gostamos e a chamamos de my_foods. Em ❷ criamos uma nova lista chamada friend_foods. Fizemos uma cópia de my_foods solicitando uma fatia de my_foods sem especificar qualquer índice e armazenamos a cópia em friend_foods. Quando exibimos cada lista, vemos que as duas contêm os mesmos alimentos: My favorite foods are: ['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are: ['pizza', 'falafel', 'carrot cake']

Para provar que realmente temos duas listas separadas, acrescentaremos um alimento em cada lista e mostraremos que cada lista mantém um registro apropriado das comidas favoritas de cada pessoa:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli') ❸ friend_foods.append('ice cream')
print("My favorite foods are:") print(my_foods)
print("\nMy friend's favorite foods are:") print(friend_foods) Em ❶, copiamos os itens originais em my_foods para a nova lista friend_foods, como fizemos no exemplo anterior. Em seguida, adicionamos um novo alimento em cada lista: em ❷, acrescentamos 'cannoli' a my_foods e em ❸, adicionamos 'ice cream' em friend_foods. Em seguida, exibimos as duas listas para ver se cada um desses alimentos está na lista apropriada.

My favorite foods are: ❹ ['pizza', 'falafel', 'carrot cake', 'cannoli']
```

```
My friend's favorite foods are: ❺ ['pizza', 'falafel', 'carrot cake',
'ice cream']
```

A saída em ❹ mostra que `'cannoli'` agora aparece em nossa lista de alimentos prediletos, mas `'ice cream'` não. Em ❺ podemos ver que `'ice cream'` agora aparece na lista de nosso amigo, mas `'cannoli'` não. Se tivéssemos simplesmente definido `friend_foods` como igual a `my_foods`, não teríamos gerado duas listas separadas. Por exemplo, eis o que acontece quando tentamos copiar uma lista sem usar uma fatia: `my_foods = ['pizza', 'falafel', 'carrot cake']`

```
# Isto não funciona: ❶ friend_foods = my_foods
my_foods.append('cannoli') friend_foods.append('ice cream')
print("My favorite foods are:") print(my_foods)
print("\nMy friend's favorite foods are:") print(friend_foods) Em vez de
armazenar uma cópia de my_foods em friend_foods em ❶, definimos que
friend_foods é igual a my_foods. Essa sintaxe, na verdade, diz a Python
para conectar a nova variável friend_foods à lista que já está em
my_foods, de modo que, agora, as duas variáveis apontam para a mesma
lista. Como resultado, quando adicionamos 'cannoli' em my_foods, essa
informação aparecerá em friend_foods. De modo semelhante, 'ice cream'
aparecerá nas duas listas, apesar de parecer que foi adicionada somente
em friend_foods.
```

A saída mostra que as duas listas agora são iguais, mas não é isso que queríamos fazer: My favorite foods are: ['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

```
My friend's favorite foods are: ['pizza', 'falafel', 'carrot cake',
'cannoli', 'ice cream']
```

NOTA Não se preocupe com os detalhes desse exemplo por enquanto. Basicamente, se você estiver tentando trabalhar com uma cópia de uma lista e vir um comportamento inesperado, certifique-se de que está copiando a lista usando uma fatia, como fizemos no primeiro exemplo.

FAÇA VOCÊ MESMO

4.10 – Fatias: Usando um dos programas que você escreveu neste capítulo, acrescente várias linhas no final do programa que façam o seguinte: • Exiba a mensagem *Os três primeiros itens da lista são:* Em seguida, use uma fatia para exibir os três primeiros itens da lista desse programa.

• Exiba a mensagem *Três itens do meio da lista são:*. Use uma fatia para exibir

três itens do meio da lista.

- Exiba a mensagem *Os três últimos itens da lista são:*. Use uma fatia para exibir os três últimos itens da lista.

4.11 – Minhas pizzas, suas pizzas: Comece com seu programa do Exercício 4.1 (página 97). Faça uma cópia da lista de pizzas e chame-a de `friend_pizzas`. Então faça o seguinte: • Adicione uma nova pizza à lista original.

- Adicione uma pizza diferente à lista `friend_pizzas`.
- Prove que você tem duas listas diferentes. Exiba a mensagem *Minhas pizzas favoritas são:; em seguida, utilize um laço* `for` *para exibir a primeira lista.* Exiba a mensagem *As pizzas favoritas de meu amigo são:*; em seguida, utilize um laço `for` para exibir a segunda lista. Certifique-se de que cada pizza nova esteja armazenada na lista apropriada.

4.12 – Mais laços: Todas as versões de *foods.py* nesta seção evitaram usar laços `for` para fazer exibições a fim de economizar espaço. Escolha uma versão de *foods.py* e escreva dois laços `for` para exibir cada lista de comidas.

Tuplas

As listas funcionam bem para armazenar conjuntos de itens que podem mudar durante a vida de um programa. A capacidade de modificar listas é particularmente importante quando trabalhamos com uma lista de usuários em um site ou com uma lista de personagens em um jogo. No entanto, às vezes, você vai querer criar uma lista de itens que não poderá mudar. As tuplas permitir fazer exatamente isso. Python refere-se a valores que não podem mudar como *imutáveis*, e uma lista imutável é chamada de *tupla*.

Definindo uma tupla

Uma tupla se parece exatamente com uma lista, exceto por usar parênteses no lugar de colchetes. Depois de definir uma tupla, podemos acessar elementos individuais usando o índice de cada item, como faríamos com uma lista.

Por exemplo, se tivermos um retângulo que sempre deva ter determinado tamanho, podemos garantir que seu tamanho não mudará colocando as dimensões em uma tupla: dimensions.py ❶ dimensions = (200, 50) ❷ print(dimensions[0]) print(dimensions[1]) Definimos a tupla `dimensions` em ❶, usando parênteses no lugar de colchetes. Em ❷ exibimos cada elemento da tupla individualmente com a mesma sintaxe que usamos para acessar elementos de uma lista: 200

```
50
```

Vamos ver o que acontece se tentarmos alterar um dos itens da tupla dimensions: `dimensions = (200, 50)` ❶ dimensions[0] = 250

O código em ❶ tenta mudar o valor da primeira dimensão, mas Python devolve um erro de tipo. Basicamente, pelo fato de estarmos tentando alterar uma tupla, o que não pode ser feito para esse tipo de objeto, Python nos informa que não podemos atribuir um novo valor a um item em uma tupla: Traceback (most recent call last): File "dimensions.py", line 3, in <module> dimensions[0] = 250

```
TypeError: 'tuple' object does not support item assignment Isso é um
ponto positivo, pois queremos que Python gere um erro se uma linha de
código tentar alterar as dimensões do retângulo.
```

Percorrendo todos os valores de uma tupla com um laço

Podemos percorrer todos os valores de uma tupla usando um laço `for`, assim como fizemos com uma lista: `dimensions = (200, 50)` for dimension in dimensions: print(dimension) Python devolve todos os elementos da tupla, como faria com uma lista: 200

```
50
```

Sobrescrevendo uma tupla

Embora não seja possível modificar uma tupla, podemos atribuir um novo valor a uma variável que armazena uma tupla. Portanto, se quiséssemos alterar nossas dimensões, poderíamos redefinir a tupla toda: ❶ `dimensions = (200, 50)` print("Original dimensions:") for dimension in dimensions: print(dimension)

```
❷ dimensions = (400, 100) ❸ print("\nModified dimensions:") for dimension
in dimensions: print(dimension) O bloco em ❶ define a tupla original e
exibe as dimensões iniciais. Em ❷ armazenamos uma nova tupla na variável
dimensions. Em seguida, exibimos as novas dimensões em ❸. Python não gera
nenhum erro dessa vez, pois sobrescrever uma variável é uma operação
válida: Original dimensions: 200
  50

  Modified dimensions: 400
  100
```

Se comparada com listas, as tuplas são estruturas de dados simples. Use-as quando quiser armazenar um conjunto de valores que não deva

ser alterado durante a vida de um programa.

FAÇA VOCÊ MESMO

4.13 – Buffet: Um restaurante do tipo buffet oferece apenas cinco tipos básicos de comida. Pense em cinco pratos simples e armazene-os em uma tupla.

- Use um laço `for` para exibir cada prato oferecido pelo restaurante.
- Tente modificar um dos itens e cerifique-se de que Python rejeita a mudança.
- O restaurante muda seu cardápio, substituindo dois dos itens com pratos diferentes. Acrescente um bloco de código que reescreva a tupla e, em seguida, use um laço `for` para exibir cada um dos itens do cardápio revisado.

Estilizando seu código

Agora que você está escrevendo programas mais longos, vale a pena conhecer algumas ideias sobre como estilizar seu código. Reserve tempo para deixar seu código o mais legível possível. Escrever códigos fáceis de ler ajuda a manter o controle sobre o que seus programas fazem e também contribui para que outras pessoas entendam seu código.

Programadores Python chegaram a um acordo sobre várias convenções de estilo a fim de garantir que o código de todos, de modo geral, esteja estruturado da mesma maneira. Depois que aprender a escrever código Python limpo, você deverá ser capaz de entender a estrutura geral do código Python de qualquer pessoa, desde que elas sigam as mesmas diretrizes. Se você espera tornar-se um programador profissional em algum momento, comece a seguir essas diretrizes o mais rápido possível, a fim de desenvolver bons hábitos.

Guia de estilo

Quando alguém quer fazer uma alteração na linguagem Python, essa pessoa escreve uma *PEP* (Python Enhancement Proposal, ou Proposta de Melhoria de Python). Uma das PEPs mais antigas é a *PEP 8*, que instrui programadores Python a estilizar seu código. A PEP é bem longa, mas boa parte dela está relacionada a estruturas de código mais complexas do que vimos até agora.

O guia de estilo de Python foi escrito considerando que o código é lido com mais frequência que é escrito. Você escreverá seu código uma

vez e então começará a lê-lo quando começar a depuração. Quando acrescentar recursos a um programa, você gastará mais tempo lendo o seu código. Ao compartilhar seu código com outros programadores, eles também o lerão.

Dada a opção entre escrever um código que seja mais fácil de escrever ou um código que seja mais fácil de ler, os programadores Python quase sempre incentivarão você a escrever um código que seja mais fácil de ler. As diretrizes a seguir ajudarão a escrever um código claro desde o início.

Indentação

A PEP 8 recomenda usar quatro espaços por nível de indentação. Usar quatro espaços melhora a legibilidade, ao mesmo tempo que deixa espaço para vários níveis de indentação em cada linha.

Em um documento de processador de texto, com frequência, as pessoas usam tabulações no lugar de espaços para indentar. Isso funciona bem para documentos de processadores de texto, mas o interpretador Python fica confuso quando tabulações são misturadas a espaços. Todo editor de texto oferece uma configuração que permite usar a tecla TAB, mas converte cada tabulação em um número definido de espaços. Definitivamente, você deve usar a tecla TAB, mas certifique-se também de que seu editor esteja configurado para inserir espaços no lugar de tabulações em seu documento.

Misturar tabulações com espaços em seu arquivo pode causar problemas que são difíceis de diagnosticar. Se você achar que tem uma mistura de tabulações e espaços, é possível converter todas as tabulações de um arquivo em espaços na maioria dos editores.

Tamanho da linha

Muitos programadores Python recomendavam que cada linha deveria ter menos de 80 caracteres. Historicamente, essa diretriz se desenvolveu porque a maioria dos computadores conseguia inserir apenas 79 caracteres em uma única linha em uma janela de terminal. Atualmente, as pessoas conseguem colocar linhas mais longas em suas telas, mas há outros motivos para se ater ao tamanho de linha-padrão de 79 caracteres. Programadores profissionais muitas vezes têm vários arquivos abertos na mesma tela, e usar o tamanho-padrão de linha lhes

permite ver linhas inteiras em dois ou três arquivos abertos lado a lado na tela. A PEP 8 também recomenda que você limite todos os seus comentários em 72 caracteres por linha, pois algumas das ferramentas que geram documentação automática para projetos maiores acrescentam caracteres de formatação no início de cada linha comentada.

As diretrizes da PEP 8 para tamanho de linha não estão gravadas a ferro e fogo, e algumas equipes preferem um limite de 99 caracteres. Não se preocupe demais com o tamanho da linha em seu código enquanto estiver aprendendo, mas saiba que as pessoas que trabalham de modo colaborativo quase sempre seguem as orientações da PEP 8. A maioria dos editores permite configurar uma indicação visual – geralmente é uma linha vertical na tela – que mostra em que ponto estão esses limites.

NOTA O Apêndice B mostra como configurar seu editor de texto de modo que ele insira quatro espaços sempre que você pressionar a tecla TAB e mostre uma linha vertical para ajudá-lo a seguir o limite de 79 caracteres.

Linhas em branco

Para agrupar partes de seu programa visualmente, utilize linhas em branco. Você deve usar linhas em branco para organizar seus arquivos, mas não as use de modo excessivo. Ao seguir os exemplos fornecidos neste livro, você deverá atingir o equilíbrio adequado. Por exemplo, se você tiver cinco linhas de código que criem uma lista e, então, outras três linhas que façam algo com essa lista, será apropriado colocar uma linha em branco entre as duas seções. Entretanto, você não deve colocar três ou quatro linhas em branco entre as duas seções.

As linhas em branco não afetarão o modo como seu código é executado, mas influenciarão em sua legibilidade. O interpretador Python utiliza indentação horizontal para interpretar o significado de seu código, mas despreza o espaçamento vertical.

Outras diretrizes de estilo

A PEP 8 tem muitas recomendações adicionais para estilo, mas a maioria das diretrizes refere-se a programas mais complexos que aqueles que você está escrevendo no momento. À medida que conhecer estruturas Python mais complexas, compartilharei as partes relevantes das

diretrizes da PEP 8.

FAÇA VOCÊ MESMO

4.14 – PEP 8: Observe o guia de estilo da PEP 8 original em *https://python.org/dev/peps/pep-0008/*. Você não usará boa parte dele agora, mas pode ser interessante dar uma olhada.

4.15 – Revisão de código: Escolha três programas que você escreveu neste capítulo e modifique-os para que estejam de acordo com a PEP 8: • Use quatro espaços para cada nível de indentação. Configure seu editor de texto para inserir quatro espaços sempre que a tecla TAB for usada, caso ainda não tenha feito isso (consulte o Apêndice B para ver instruções sobre como fazê-lo).

- Use menos de 80 caracteres em cada linha e configure seu editor para que mostre uma linha vertical na posição do caractere de número 80.
- Não use linhas em branco em demasia em seus arquivos de programa.

Resumo

Neste capítulo aprendemos a trabalhar de modo eficiente com os elementos de uma lista. Vimos como percorrer uma lista usando um laço `for`, como Python usa indentação para estruturar um programa e como evitar alguns erros comuns de indentação. Aprendemos a criar listas numéricas simples, além de conhecermos algumas operações que podem ser realizadas em listas numéricas. Aprendemos a fatiar uma lista para trabalhar com um subconjunto de itens e a copiar listas de modo adequado usando uma fatia. Também conhecemos as tuplas, que oferecem um nível de proteção a um conjunto de valores que não deve ser alterado, e aprendemos a estilizar seu código cada vez mais complexo para facilitar sua leitura.

No Capítulo 5 veremos como responder de forma apropriada a diferentes condições usando instruções `if`. Aprenderemos a combinar conjuntos relativamente complexos de testes condicionais para responder de forma apropriada ao tipo de situação desejada ou oferecer informações que você esteja procurando. Também aprenderemos a usar instruções `if` enquanto percorremos uma lista para realizar ações específicas com elementos selecionados de uma lista.