5
Instruções if

Com frequência, a programação envolve analisar um conjunto de condições e decidir qual ação deve ser executada de acordo com essas condições. A instrução `if` de Python permite analisar o estado atual de um programa e responder de forma apropriada a esse estado.

Neste capítulo aprenderemos a escrever testes condicionais, que permitem verificar qualquer condição que seja de seu interesse. Veremos como escrever instruções `if` simples e criar uma série de instruções `if` mais complexa para identificar se as condições exatas que você quer estão presentes. Então você aplicará esse conceito a listas, de modo que será possível escrever um laço `for` que trate a maioria dos itens de uma lista de uma maneira, mas determinados itens com valores específicos de modo diferente.

Um exemplo simples

O pequeno exemplo a seguir mostra como testes `if` permitem responder a situações especiais de forma correta. Suponha que você tenha uma lista de carros e queira exibir o nome de cada carro. Carros têm nomes próprios, portanto a maioria deles deve ser exibido com a primeira letra maiúscula. No entanto, o valor `'bmw'` deve ser apresentado somente com letras maiúsculas. O código a seguir percorre uma lista de nomes de carros em um laço e procura o valor `'bmw'`. Sempre que o valor for `'bmw'`, ele será exibido com letras maiúsculas, e não apenas com a inicial maiúscula: cars.py cars = ['audi', 'bmw', 'subaru', 'toyota']

```
for car in cars: ❶ if car == 'bmw': print(car.upper()) else:
print(car.title()) O laço nesse exemplo inicialmente verifica se o valor
atual de car é 'bmw' ❶. Em caso afirmativo, o valor é exibido com letras
maiúsculas. Se o valor de car for diferente de 'bmw', será exibido com a
primeira letra maiúscula: Audi
BMW
Subaru
Toyota
```

Esse exemplo combina vários conceitos que veremos neste capítulo. Vamos começar observando os tipos de teste que podemos usar para analisar as condições em seu programa.

Testes condicionais

No coração de cada instrução `if` está uma expressão que pode ser avaliada como `True` ou `False`, chamada de *teste condicional*. Python usa os valores `True` e `False` para decidir se o código em uma instrução `if` deve ser executado. Se um teste condicional for avaliado como `True`, Python executará o código após a instrução `if`. Se o teste for avaliado como `False`, o interpretador ignorará o código depois da instrução `if`.

Verificando a igualdade

A maioria dos testes condicionais compara o valor atual de uma variável com um valor específico de interesse. O teste condicional mais simples verifica se o valor de uma variável é igual ao valor de interesse: ❶

```
>>> car = 'bmw'
❷ >>> car == 'bmw'
  True
```

A linha em ❶ define o valor de `car` como `'bmw'` usando um único sinal de igualdade, como já vimos muitas vezes. A linha em ❷ verifica se o valor de `car` é `'bmw'` usando um sinal de igualdade duplo (`==`). Esse *operador de igualdade* devolve `True` se os valores dos lados esquerdo e direito do operador forem iguais, e `False` se forem diferentes. Os valores nesse exemplo são iguais, portanto Python devolve `True`.

Quando o valor de `car` for diferente de `'bmw'`, esse teste devolverá `False`:

```
❶ >>> car = 'audi'
❷ >>> car == 'bmw'
  False
```

Um único sinal de igual, na verdade, é uma instrução; você poderia ler o código em ❶ como "defina o valor de `car` como `'audi'`". Por outro lado, um sinal de igualdade duplo, como o que está em ❷, faz uma pergunta: "O valor de `car` é igual a `'bmw'`?". A maioria das linguagens de programação utiliza sinais de igualdade dessa maneira.

Ignorando as diferenças entre letras maiúsculas e minúsculas ao verificar a igualdade

Testes de igualdade diferenciam letras maiúsculas de minúsculas em Python. Por exemplo, dois valores com diferenças apenas quanto a letras maiúsculas ou minúsculas não são considerados iguais: >>> **`car = 'Audi'`**

```
>>> car == 'audi'
False
```

Se a diferença entre letras maiúsculas e minúsculas for importante, esse comportamento será vantajoso. Porém, se essa diferença não importar e você simplesmente quiser testar o valor de uma variável, poderá converter esse valor para letras minúsculas antes de fazer a comparação: >>> **`car = 'Audi'`**

```
>>> car.lower() == 'audi'
True
```

Esse teste deve devolver `True`, independentemente do modo como o valor `'Audi'` estiver formatado, pois o teste agora ignora as diferenças entre letras maiúsculas e minúsculas. A função `lower()` não altera o valor originalmente armazenado em `car`, portanto você pode fazer esse tipo de comparação sem afetá-lo: ❶ >>> **`car = 'Audi'`**

```
❷ >>> car.lower() == 'audi'
  True
❸ >>> car 'Audi'
```

Em ❶ armazenamos a string `'Audi'` com a primeira letra maiúscula na variável `car`. Em ❷ convertemos o valor de `car` para letras minúsculas e comparamos esse valor com a string `'audi'`. As duas strings são iguais, portanto Python devolve `True`. Em ❸ podemos ver que o valor armazenado em `car` não foi afetado pelo teste condicional.

Os sites impõem determinadas regras para os dados fornecidos pelos usuários de modo semelhante a esse. Por exemplo, um site pode usar um teste condicional desse tipo para garantir que todos os usuários tenham um nome realmente único, e não apenas uma variação quanto ao uso de letras maiúsculas em relação ao nome de usuário de outra pessoa. Quando alguém submeter um novo nome de usuário, esse nome será convertido para letras minúsculas e comparado às versões com letras minúsculas de todos os nomes de usuário existentes. Durante

essa verificação, um nome de usuário como `'John'` será rejeitado se houver qualquer variação de `'john'` já em uso.

Verificando a diferença

Se quiser determinar se dois valores não são iguais, você poderá combinar um ponto de exclamação e um sinal de igualdade (`!=`). O ponto de exclamação representa *não*, como ocorre em muitas linguagens de programação.

Vamos usar outra instrução `if` para ver como o operador "diferente de" é usado. Armazenaremos o ingrediente pedido em uma pizza em uma variável e então exibiremos uma mensagem se a pessoa não pediu anchovas: toppings.py requested_topping = 'mushrooms'

```
❶ if requested_topping != 'anchovies': print("Hold the anchovies!") A
linha em ❶ compara o valor de requested_topping com o valor 'anchovies'.
Se esses dois valores não forem iguais, Python devolverá True e executará o
código após a instrução if. Se esses dois valores forem iguais, Python
devolverá False e não executará o código após essa instrução.
```

Como o valor de `requested_topping` não é `'anchovies'`, a instrução `print` é executada: Hold the anchovies!

A maior parte das expressões condicionais que você escrever testará a igualdade, porém, às vezes, você achará mais eficiente testar a não igualdade.

Comparações numéricas

Testar valores numéricos é bem simples. Por exemplo, o código a seguir verifica se uma pessoa tem 18 anos: >>> **`age = 18`**

```
>>> age == 18
True
```

Também podemos testar se dois números não são iguais. Por exemplo, o código a seguir exibe uma mensagem se a resposta dada não estiver correta: magic_number.py answer = 17

```
❶ if answer != 42: print("That is not the correct answer. Please try
again!") O teste condicional em ❶ passa porque o valor de answer (17) não
é igual a 42. Como o teste passa, o bloco de código indentado é executado:
That is not the correct answer. Please try again!
```

Você pode incluir também várias comparações matemáticas em suas instruções condicionais, por exemplo, menor que, menor ou igual a, maior que e maior ou igual a: >>> **`age = 19`**

```
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Toda comparação matemática pode ser usada como parte de uma instrução `if`, o que pode ajudar você a determinar as condições exatas de interesse.

Testando várias condições

Pode ser que você queira testar várias condições ao mesmo tempo. Por exemplo, ocasionalmente, você pode precisar de duas condições `True` para executar uma ação. Em outros casos, poderia ficar satisfeito com apenas uma condição `True`. As palavras reservadas `and` e `or` podem ajudar nessas situações.

Usando and para testar várias condições

Para verificar se duas condições são `True` simultaneamente, utilize a palavra reservada `and` para combinar os dois testes condicionais; se cada um dos testes passar, a expressão como um todo será avaliada como `True`. Se um dos testes falhar, ou ambos, a expressão será avaliada como `False`.

Por exemplo, podemos verificar se duas pessoas têm mais de 21 anos usando o teste a seguir: ❶ >>> **`age_0 = 22`**

```
>>> age_1 = 18
❷ >>> age_0 >= 21 and age_1 >= 21
False
❸ >>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True
```

Em ❶ definimos duas idades, `age_0` e `age_1`. Em ❷ verificamos se as duas idades são maiores ou iguais a 21. O teste à esquerda passa,

porém, o teste à direita falha, portanto a expressão condicional como um todo é avaliada como `False`. Em ❸ mudamos `age_1` para 22. O valor de `age_1` agora é maior que 21, portanto os dois testes individuais passam, fazendo com que a expressão condicional como um todo seja avaliada como `True`.

Para melhorar a legibilidade, podemos usar parênteses em torno dos testes individuais, mas eles não são obrigatórios. Se usar parênteses, seu teste terá o seguinte aspecto: (age_0 >= 21) and (age_1 >= 21)

Usando or para testar várias condições

A palavra reservada `or` permite verificar várias condições também, mas o teste passa se um dos testes individuais passar, ou ambos. Uma expressão `or` falha somente quando os dois testes individuais falharem.

Vamos considerar duas idades novamente, porém, desta vez, queremos que apenas uma das pessoas tenha mais de 21 anos: ❶

```
>>> age_0 = 22
>>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
True
❸ >>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Começamos novamente com duas variáveis para idade em ❶. Como o teste para `age_0` em ❷ passa, a expressão com um todo é avaliada como `True`. Então diminuímos `age_0` para 18. Em ❸ os dois testes agora falham e a expressão como um todo é avaliada como `False`.

Verificando se um valor está em uma lista

Às vezes, é importante verificar se uma lista contém um determinado valor antes de executar uma ação. Por exemplo, talvez você queira verificar se um novo nome de usuário já existe em uma lista de nomes de usuários atuais antes de concluir o registro de alguém em um site. Em um projeto de mapeamento, você pode querer verificar se uma localidade submetida já existe em uma lista de localidades conhecidas.

Para descobrir se um valor em particular já está em uma lista, utilize a palavra reservada `in`. Vamos observar um código que você poderia escrever para uma pizzaria. Criaremos uma lista de ingredientes que um

cliente pediu em sua pizza e, então, verificaremos se determinados ingredientes estão na lista.

```
  >>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
❶ >>> 'mushrooms' in requested_toppings True
❷ >>> 'pepperoni' in requested_toppings False
```

Em ❶ e em ❷, a palavra reservada `in` diz a Python para verificar a existência de `'mushrooms'` e de `'pepperoni'` na lista `requested_toppings`. Essa técnica é bem eficaz, pois podemos criar uma lista de valores essenciais e verificar facilmente se o valor que estamos testando é igual a um dos valores da lista.

Verificando se um valor não está em uma lista

Em outras ocasiões, é importante saber se um valor não está em uma lista. Podemos usar a palavra reservada `not` nesse caso. Por exemplo, considere uma lista de usuários que foi impedida de fazer comentários em um fórum. Podemos verificar se um usuário foi banido antes de permitir que essa pessoa submeta um comentário: banned_users.py banned_users = ['andrew', 'carolina', 'david']

```
  user = 'marie'


❶ if user not in banned_users: print(user.title() + ", you can post a
response if you wish.") A linha em ❶ é bem clara. Se o valor de user não
estiver na lista banned_users, Python devolverá True e executará a linha
indentada.
```

A usuária `'marie'` não está na lista `banned_users`, portanto ela vê uma mensagem convidando-a a postar uma resposta: Marie, you can post a response if you wish.

Expressões booleanas

À medida que aprender mais sobre programação, você ouvirá o termo *expressão booleana* em algum momento. Uma expressão booleana é apenas outro nome para um teste condicional. Um *valor booleano* é `True` ou `False`, exatamente como o valor de uma expressão condicional após ter sido avaliada.

Valores booleanos muitas vezes são usados para manter o controle de determinadas condições, como o fato de um jogo estar executando ou

se um usuário pode editar certos conteúdos em um site: game_active = True can_edit = False Valores booleanos oferecem uma maneira eficiente para monitorar o estado ou uma condição em particular que seja importante para o seu programa.

FAÇA VOCÊ MESMO

5.1 – Testes condicionais: Escreva uma série de testes condicionais. Exiba uma frase que descreva o teste e o resultado previsto para cada um. Seu código deverá ser semelhante a: car = 'subaru'
`print("Is car == 'subaru'? I predict True.") print(car == 'subaru')`
`print("\nIs car == 'audi'? I predict False.") print(car == 'audi')` • `Observe atentamente seus resultados e certifique-se de que compreende por que cada linha é avaliada como True ou False.`

- Crie pelo menos dez testes. Tenha no mínimo cinco testes avaliados como `True` e outros cinco avaliados como `False`.

5.2 – Mais testes condicionais: Você não precisa limitar o número de testes que criar em dez. Se quiser testar mais comparações, escreva outros testes e acrescente-os em *conditional_tests.py*. Tenha pelo menos um resultado `True` e um `False` para cada um dos casos a seguir: • testes de igualdade e de não igualdade com strings; • testes usando a função `lower()`; • testes numéricos que envolvam igualdade e não igualdade, maior e menor que, maior ou igual a e menor ou igual a; • testes usando as palavras reservadas `and` e `or`; • testes para verificar se um item está em uma lista; • testes para verificar se um item não está em uma lista.

Instruções if

Quando compreender os testes condicionais, você poderá começar a escrever instruções `if`. Há vários tipos de instruções `if`, e a escolha de qual deles usar dependerá do número de condições que devem ser testadas. Vimos vários exemplos de instruções `if` na discussão sobre testes condicionais, mas agora vamos explorar mais o assunto.

Instruções if simples

O tipo mais simples de instrução `if` tem um teste e uma ação: `if` *`teste_condicional`*`:` *`faça algo`* Você pode colocar qualquer teste condicional na primeira linha, e praticamente qualquer ação no bloco indentado após o teste. Se o teste condicional for avaliado como `True`, Python executará o código após a instrução `if`. Se o teste for avaliado como `False`, Python ignorará o código depois da instrução `if`.

Suponha que temos uma variável que representa a idade de uma pessoa e queremos saber se essa pessoa tem idade suficiente para votar. O código a seguir testa se a pessoa pode votar: voting.py age = 19
❶ if age >= 18: ❷ print("You are old enough to vote!") Em ❶ Python verifica se o valor em age é maior ou igual a 18. É maior, portanto a instrução indentada print em ❷ é executada: You are old enough to vote!

A indentação tem a mesma função em instruções `if` que aquela desempenhada em laços `for`. Todas as linhas indentadas após uma instrução `if` serão executadas se o teste passar, e todo o bloco de linhas indentadas será ignorado se o teste não passar.

Podemos ter tantas linhas de código quantas quisermos no bloco após a instrução `if`. Vamos acrescentar outra linha de saída se a pessoa tiver idade suficiente para votar, perguntando se o indivíduo já se registrou para votar: age = 19

```
if age >= 18: print("You are old enough to vote!") print("Have you
registered to vote yet?") O teste condicional passa e as duas instruções
print estão indentadas, portanto ambas as linhas são exibidas: You are
old enough to vote!
Have you registered to vote yet?
```

Se o valor de `age` for menor que 18, esse programa não apresentará nenhuma saída.

Instruções if-else

Com frequência você vai querer executar uma ação quando um teste condicional passar, e uma ação diferente em todos os demais casos. A sintaxe `if-else` de Python torna isso possível. Um bloco `if-else` é semelhante a uma instrução `if` simples, porém a instrução `else` permite definir uma ação ou um conjunto de ações executado quando o teste condicional falhar.

Exibiremos a mesma mensagem mostrada antes se a pessoa tiver idade suficiente para votar, porém, desta vez, acrescentaremos uma mensagem para qualquer um que não tenha idade suficiente para votar: age = 17
❶ if age >= 18: print("You are old enough to vote!") print("Have you registered to vote yet?") ❷ else: print("Sorry, you are too young to vote.") print("Please register to vote as soon as you turn 18!") Se o teste condicional em ❶ passar, o primeiro bloco de instruções print indentadas será executado. Se o teste for avaliado como False, o bloco else em ❷ será executado. Como age é menor que 18 dessa vez, o teste condicional falha e o código do bloco else é executado: Sorry, you are too young to vote.

```
Please register to vote as soon as you turn 18!
```

Esse código funciona porque há apenas duas possíveis situações para avaliar: uma pessoa tem idade suficiente para votar ou não tem. A estrutura `if-else` funciona bem em situações em que você quer que Python sempre execute uma de duas possíveis ações. Em uma cadeia `if-else` simples como essa, uma das duas ações sempre será executada.

Sintaxe if-elif-else

Muitas vezes, você precisará testar mais de duas situações possíveis; para avaliar isso, a sintaxe `if-elif-else` de Python poderá ser usada. Python executa apenas um bloco em uma cadeia `if-elif-else`. Cada teste condicional é executado em sequência, até que um deles passe. Quando um teste passar, o código após esse teste será executado e Python ignorará o restante dos testes.

Muitas situações do mundo real envolvem mais de duas condições possíveis. Por exemplo, considere um parque de diversões que cobre preços distintos para grupos etários diferentes: • a entrada para qualquer pessoa com menos de 4 anos é gratuita; • a entrada para qualquer pessoa com idade entre 4 e 18 anos custa 5 dólares; • a entrada para qualquer pessoa com 18 anos ou mais custa 10 dólares.

Como podemos usar uma instrução `if` para determinar o preço da entrada de alguém? O código a seguir testa a faixa etária de uma pessoa e então exibe uma mensagem com o preço da entrada: amusement_park.py age = 12

```
❶ if age < 4: print("Your admission cost is $0.") ❷ elif age < 18:
print("Your admission cost is $5.") ❸ else: print("Your admission cost is
$10.") O teste if em ❶ verifica se uma pessoa tem menos de 4 anos. Se o
teste passar, uma mensagem apropriada será exibida e Python ignorará o
restante dos testes. A linha elif em ❷, na verdade, é outro teste if,
executado somente se o teste anterior falhar. Nesse ponto da cadeia,
sabemos que a pessoa tem pelo menos 4 anos de idade, pois o primeiro teste
falhou. Se a pessoa tiver menos de 18 anos, uma mensagem apropriada será
exibida, e Python ignorará o bloco else. Se tanto o teste em if quanto o
teste em elif falharem, Python executará o código do bloco else em ❸.
```

Nesse exemplo, o teste em ❶ é avaliado como `False`, portanto seu bloco de código não é executado. No entanto, o segundo teste é

avaliado como `True` (12 é menor que 18), portanto seu código é executado. A saída é uma frase que informa o preço da entrada ao usuário: Your admission cost is $5.

Qualquer idade acima de 17 anos faria os dois primeiros testes falharem. Nessas situações, o bloco `else` seria executado e o preço da entrada seria de 10 dólares.

Em vez de exibir o preço da entrada no bloco `if-elif-else`, seria mais conciso apenas definir o preço na cadeia `if-elif-else` e, então, ter uma instrução `print` simples que execute depois que a cadeia for avaliada: `age = 12`

```
if age < 4: ❶ price = 0
elif age < 18: ❷ price = 5
else: ❸ price = 10

❹ print("Your admission cost is $" + str(price) + ".") As linhas em ❶, ❷ e ❸ definem o valor de price conforme a idade da pessoa, como no exemplo anterior. Depois que o preço é definido pela cadeia if-elif-else, uma instrução print separada e não indentada ❹ utiliza esse valor para exibir uma mensagem informando o preço de entrada da pessoa.
```

Esse código gera a mesma saída do exemplo anterior, mas o propósito da cadeia `if-elif-else` é mais restrito. Em vez de determinar um preço e exibir uma mensagem, ela simplesmente determina o preço da entrada. Além de ser mais eficiente, esse código revisado é mais fácil de modificar que a abordagem original. Para mudar o texto da mensagem de saída, seria necessário alterar apenas uma instrução `print`, e não três instruções `print` separadas.

Usando vários blocos elif

Podemos usar quantos blocos `elif` quisermos em nosso código. Por exemplo, se o parque de diversões implementasse um desconto para idosos, você poderia acrescentar mais um teste condicional no código a fim de determinar se uma pessoa está qualificada a receber esse desconto. Suponha que qualquer pessoa com 65 anos ou mais pague metade do preço normal da entrada, isto é, 5 dólares: `age = 12`

```
if age < 4: price = 0
elif age < 18: price = 5
```

```
❶ elif age < 65: price = 10
❷ else: price = 5

  print("Your admission cost is $" + str(price) + ".") A maior parte desse
  código não foi alterada. O segundo bloco elif em ❶ agora faz uma
  verificação para garantir que uma pessoa tenha menos de 65 anos antes de
  lhe cobrar o preço da entrada inteira, que é de 10 dólares. Observe que o
  valor atribuído no bloco else em ❷ precisa ser alterado para 5 dólares,
  pois as únicas idades que chegam até esse bloco são de pessoas com 65
  anos ou mais.
```

Omitindo o bloco else

Python não exige um bloco `else` no final de uma cadeia `if-elif`. Às vezes, um bloco `else` é útil; outras vezes, é mais claro usar uma instrução `elif` adicional que capture a condição específica de interesse: `age = 12`

```
  if age < 4: price = 0
  elif age < 18: price = 5
  elif age < 65: price = 10
❶ elif age >= 65: price = 5

  print("Your admission cost is $" + str(price) + ".") O bloco elif extra
  em ❶ atribui um preço de 5 dólares quando a pessoa tiver 65 anos ou
  mais, o que é um pouco mais claro que o bloco else geral. Com essa
  mudança, todo bloco de código deve passar por um teste específico para
  ser executado.
```

O bloco `else` é uma instrução que captura tudo. Ela corresponde a qualquer condição não atendida por um teste `if` ou `elif` específicos e isso, às vezes, pode incluir dados inválidos ou até mesmo maliciosos. Se você tiver uma condição final específica para testar, considere usar um último bloco `elif` e omitir o bloco `else`. Como resultado, você terá mais confiança de que seu código executará somente nas condições corretas.

Testando várias condições

A cadeia `if-elif-else` é eficaz, mas é apropriada somente quando você quiser que apenas um teste passe. Assim que encontrar um teste que passe, o interpretador Python ignorará o restante dos testes. Esse comportamento é vantajoso, pois é eficiente e permite testar uma condição específica.

Às vezes, porém, é importante verificar todas as condições de interesse. Nesse caso, você deve usar uma série de instruções `if` simples, sem blocos `elif` ou `else`. Essa técnica faz sentido quando mais de uma condição pode ser `True` e você quer atuar em todas as condições que sejam verdadeiras.

Vamos reconsiderar o exemplo da pizzaria. Se alguém pedir uma pizza com dois ingredientes, será necessário garantir que esses dois ingredientes sejam incluídos em sua pizza: toppings.py ❶ requested_toppings = ['mushrooms', 'extra cheese']

```
❷ if 'mushrooms' in requested_toppings: print("Adding mushrooms.") ❸ if
'pepperoni' in requested_toppings: print("Adding pepperoni.") ❹ if 'extra
cheese' in requested_toppings: print("Adding extra cheese.")
  print("\nFinished making your pizza!") Começamos em ❶ com uma lista
  contendo os ingredientes solicitados. A instrução if em ❷ verifica se a
  pessoa pediu cogumelos ('mushrooms') em sua pizza. Em caso afirmativo,
  uma mensagem será exibida para confirmar esse ingrediente. O teste para
  pepperoni em ❸ corresponde a outra instrução if simples, e não a uma
  instrução elif ou else, portanto esse teste é executado independentemente
  de o teste anterior ter passado ou não. O código em ❹ verifica se queijo
  extra ('extra cheese') foi pedido, não importando o resultado dos dois
  primeiros testes. Esses três testes independentes são realizados sempre
  que o programa é executado.
```

Como todas as condições nesse exemplo são avaliadas, tanto cogumelos quanto queijo extra são adicionados à pizza: Adding mushrooms.

```
  Adding extra cheese.


  Finished making your pizza!
```

Esse código não funcionaria de modo apropriado se tivéssemos usado um bloco `if-elif-else`, pois o código pararia de executar depois que apenas um teste tivesse passado. Esse código seria assim: `requested_toppings = ['mushrooms', 'extra cheese']`

```
  if 'mushrooms' in requested_toppings: print("Adding mushrooms.") elif
  'pepperoni' in requested_toppings: print("Adding pepperoni.") elif 'extra
  cheese' in requested_toppings: print("Adding extra cheese.")
```

```
print("\nFinished making your pizza!") O teste para 'mushrooms' é o
primeiro a passar, portanto cogumelos são adicionados à pizza. No
entanto, os valores 'extra cheese' e 'pepperoni' não são verificados,
pois Python não executa nenhum teste depois do primeiro que passar em uma
cadeia if-elif-else. O primeiro ingrediente do cliente será adicionado,
mas todos os demais não serão: Adding mushrooms.


Finished making your pizza!
```

Em suma, se quiser que apenas um bloco de código seja executado, utilize uma cadeia `if-elif-else`. Se mais de um bloco de código deve executar, utilize uma série de instruções `if` independentes.

FAÇA VOCÊ MESMO

5.3 – Cores de alienígenas #1: Suponha que um alienígena acabou de ser atingido em um jogo. Crie uma variável chamada `alien_color` e atribua-lhe um valor igual a `'green'`, `'yellow'` ou `'red'`.

- Escreva uma instrução `if` para testar se a cor do alienígena é verde. Se for, mostre uma mensagem informando que o jogador acabou de ganhar cinco pontos.
- Escreva uma versão desse programa em que o teste `if` passe e outro em que ele falhe. (A versão que falha não terá nenhuma saída.) **5.4 – Cores de alienígenas #2:** Escolha uma cor para um alienígena, como foi feito no Exercício 5.3, e escreva uma cadeia `if-else`.
- Se a cor do alienígena for verde, mostre uma frase informando que o jogador acabou de ganhar cinco pontos por atingir o alienígena.
- Se a cor do alienígena não for verde, mostre uma frase informando que o jogador acabou de ganhar dez pontos.
- Escreva uma versão desse programa que execute o bloco `if` e outro que execute o bloco `else`.

5.5 – Cores de alienígenas #3: Transforme sua cadeia `if-else` do Exercício 5.4 em uma cadeia `if-elif-else`.

- Se o alienígena for verde, mostre uma mensagem informando que o jogador ganhou cinco pontos.
- Se o alienígena for amarelo, mostre uma mensagem informando que o jogador ganhou dez pontos.
- Se o alienígena for vermelho, mostre uma mensagem informando que o jogador ganhou quinze pontos.
- Escreva três versões desse programa, garantindo que cada mensagem seja exibida para a cor apropriada do alienígena.

5.6 – Estágios da vida: Escreva uma cadeia `if-elif-else` que determine o

estágio da vida de uma pessoa. Defina um valor para a variável **age** e então: • Se a pessoa tiver menos de 2 anos de idade, mostre uma mensagem dizendo que ela é um bebê.

- Se a pessoa tiver pelo menos 2 anos, mas menos de 4, mostre uma mensagem dizendo que ela é uma criança.
- Se a pessoa tiver pelo menos 4 anos, mas menos de 13, mostre uma mensagem dizendo que ela é um(a) garoto(a).
- Se a pessoa tiver pelo menos 13 anos, mas menos de 20, mostre uma mensagem dizendo que ela é um(a) adolescente.
- Se a pessoa tiver pelo menos 20 anos, mas menos de 65, mostre uma mensagem dizendo que ela é adulto.
- Se a pessoa tiver 65 anos ou mais, mostre uma mensagem dizendo que essa pessoa é idoso.

5.7 – Fruta favorita: Faça uma lista de suas frutas favoritas e, então, escreva uma série de instruções `if` independentes que verifiquem se determinadas frutas estão em sua lista.

- Crie uma lista com suas três frutas favoritas e chame-a de `favorite_fruits`.
- Escreva cinco instruções `if`. Cada instrução deve verificar se uma determinada fruta está em sua lista. Se estiver, o bloco `if` deverá exibir uma frase, por exemplo, *Você realmente gosta de bananas!*

Usando instruções if com listas

Algumas tarefas interessantes podem ser feitas se combinarmos listas com instruções `if`. Podemos prestar atenção em valores especiais, que devam ser tratados de modo diferente de outros valores da lista. Podemos administrar mudanças de condições de modo eficiente, por exemplo, a disponibilidade de determinados itens em um restaurante durante um turno. Também podemos começar a provar que nosso código funciona conforme esperado em todas as possíveis situações.

Verificando itens especiais

Este capítulo começou com um exemplo simples mostrando como lidar com um valor especial como `'bmw'`, que devia ser exibido em um formato diferente de outros valores da lista. Agora que você tem uma compreensão básica a respeito dos testes condicionais e de instruções `if`, vamos observar com mais detalhes de que modo podemos prestar atenção em valores especiais de uma lista e tratá-los de forma apropriada.

Vamos prosseguir com o exemplo da pizzaria. A pizzaria exibe uma mensagem sempre que um ingrediente é adicionado à sua pizza à medida que ela é preparada. O código para essa ação pode ser escrito de modo bem eficiente se criarmos uma lista de ingredientes solicitados pelo cliente e usarmos um laço para anunciar cada ingrediente à medida que ele é acrescentado à pizza: toppings.py requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

```
for requested_topping in requested_toppings: print("Adding " +
requested_topping + ".")
print("\nFinished making your pizza!") A saída é simples, pois esse
código é composto apenas de um laço for simples: Adding mushrooms.

Adding green peppers.
Adding extra cheese.


Finished making your pizza!
```

E se a pizzaria ficasse sem pimentões verdes? Uma instrução `if` no laço `for` pode tratar essa situação de modo apropriado: `requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']`

```
for requested_topping in requested_toppings: ❶ if requested_topping ==
'green peppers': print("Sorry, we are out of green peppers right now.")
❷ else: print("Adding " + requested_topping + ".")
print("\nFinished making your pizza!") Dessa vez verificamos cada item
solicitado antes de adicioná-lo à pizza. O código em ❶ verifica se a
pessoa pediu pimentões verdes. Em caso afirmativo, exibimos uma mensagem
informando por que ela não pode ter pimentões verdes. O bloco else em ❷
garante que todos os demais ingredientes serão adicionados à pizza.
```

A saída mostra que cada ingrediente solicitado é tratado de forma apropriada.

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.


Finished making your pizza!
```

Verificando se uma lista não está vazia

Fizemos uma suposição simples sobre todas as listas com as quais trabalhamos até agora: supomos que toda lista continha pelo menos um item. Em breve, deixaremos os usuários fornecerem as informações a serem armazenadas em uma lista, portanto não poderemos supor que uma lista contenha um item sempre que um laço for executado. Nessa situação, será conveniente testar se uma lista está vazia antes de executar um laço `for`.

Como exemplo, vamos verificar se a lista de ingredientes solicitados está vazia antes de prepararmos a pizza. Se a lista estiver vazia, perguntaremos ao usuário se ele realmente quer uma pizza simples. Se a lista não estiver vazia, prepararemos a pizza exatamente como fizemos nos exemplos anteriores: ❶ requested_toppings = []

```
❷ if requested_toppings: for requested_topping in requested_toppings:
print("Adding " + requested_topping + ".") print("\nFinished making your
pizza!") ❸ else: print("Are you sure you want a plain pizza?") Dessa vez,
começamos com uma lista vazia de ingredientes solicitados em ❶. Em vez de
passar diretamente para um laço for, fazemos uma verificação rápida em ❷.
Quando o nome de uma lista é usado em uma instrução if, Python devolve True
se a lista contiver pelo menos um item; uma lista vazia é avaliada como
False. Se requested_toppings passar no teste condicional, executamos o
mesmo laço for do exemplo anterior. Se o teste condicional falhar, exibimos
uma mensagem perguntando ao cliente se ele realmente quer uma pizza
simples, sem ingredientes adicionais ❸.
```

Nesse caso, a lista está vazia, portanto a saída contém uma pergunta para saber se o usuário realmente quer uma pizza simples: Are you sure you want a plain pizza?

Se a lista não estiver vazia, a saída mostrará cada ingrediente solicitado adicionado à pizza.

Usando várias listas

As pessoas pedirão de tudo, em especial quando se tratar de ingredientes para uma pizza. E se um cliente realmente quiser batatas fritas em sua pizza? Podemos usar listas e instruções `if` para garantir que o dado de entrada faça sentido antes de atuar sobre ele.

Vamos prestar atenção em solicitações de ingredientes incomuns antes de prepararmos uma pizza. O exemplo a seguir define duas listas. A primeira é uma lista de ingredientes disponíveis na pizzaria, e a segunda

é a lista de ingredientes que o usuário pediu. Dessa vez, cada item em `requested_toppings` é verificado em relação à lista de ingredientes disponíveis antes de ser adicionado à pizza: ❶ available_toppings = ['mushrooms', 'olives', 'green peppers', 'pepperoni', 'pineapple', 'extra cheese']

```
❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings: ❹ if requested_topping in
available_toppings: print("Adding " + requested_topping + ".") ❺ else:
print("Sorry, we don't have " + requested_topping + ".")
  print("\nFinished making your pizza!") Em ❶ definimos uma lista de
  ingredientes disponíveis nessa pizzaria. Observe que essa informação
  poderia ser uma tupla se a pizzaria tiver uma seleção estável de
  ingredientes. Em ❷ criamos uma lista de ingredientes solicitados por um
  cliente. Observe a solicitação incomum, 'french fries' (batatas fritas).
  Em ❸ percorremos a lista de ingredientes solicitados em um laço. Nesse
  laço, inicialmente verificamos se cada ingrediente solicitado está na
  lista de ingredientes disponíveis ❹. Se estiver, adicionamos esse
  ingrediente na pizza. Se o ingrediente solicitado não estiver na lista de
  ingredientes disponíveis, o bloco else será executado ❺. Esse bloco
  exibe uma mensagem informando ao usuário quais ingredientes não estão
  disponíveis.
```

A sintaxe desse código gera uma saída clara e informativa: Adding mushrooms.

```
  Sorry, we don't have french fries.
  Adding extra cheese.


  Finished making your pizza!
```

Com apenas algumas linhas de código, conseguimos tratar uma situação do mundo real de modo bem eficiente!

FAÇA VOCÊ MESMO

5.8 – Olá admin: Crie uma lista com cinco ou mais nomes de usuários, incluindo o nome `'admin'`. Suponha que você esteja escrevendo um código que exibirá uma saudação a cada usuário depois que eles fizerem login em um site. Percorra a lista com um laço e mostre uma saudação para cada usuário: • Se o nome do usuário for `'admin'`, mostre uma saudação especial, por exemplo, *Olá admin, gostaria de ver um relatório de status?*

- Caso contrário, mostre uma saudação genérica, como *Olá Eric, obrigado por fazer login novamente*.

5.9 – Sem usuários: Acrescente um teste `if` em *hello_admin.py* para garantir que a lista de usuários não esteja vazia.

- Se a lista estiver vazia, mostre a mensagem *Precisamos encontrar alguns usuários!*
- Remova todos os nomes de usuário de sua lista e certifique-se de que a mensagem correta seja exibida.

5.10 – Verificando nomes de usuários: Faça o seguinte para criar um programa que simule o modo como os sites garantem que todos tenham um nome de usuário único.

- Crie uma lista chamada `current_users` com cinco ou mais nomes de usuários.
- Crie outra lista chamada `new_users` com cinco nomes de usuários. Garanta que um ou dois dos novos usuários também estejam na lista `current_users`.
- Percorra a lista `new_users` com um laço para ver se cada novo nome de usuário já foi usado. Em caso afirmativo, mostre uma mensagem informando que a pessoa deverá fornecer um novo nome. Se um nome de usuário não foi usado, apresente uma mensagem dizendo que o nome do usuário está disponível.
- Certifique-se de que sua comparação não levará em conta as diferenças entre letras maiúsculas e minúsculas. Se `'John'` foi usado, `'JOHN'` não deverá ser aceito.

5.11 – Números ordinais: Números ordinais indicam sua posição em uma lista, por exemplo, *1st* ou *2nd*, em inglês. A maioria dos números ordinais nessa língua termina com *th*, exceto 1, 2 e 3.

- Armazene os números de 1 a 9 em uma lista.
- Percorra a lista com um laço.
- Use uma cadeia `if-elif-else` no laço para exibir a terminação apropriada para cada número ordinal. Sua saída deverá conter `"1st 2nd 3rd 4th 5th 6th 7th 8th 9th"`, e cada resultado deve estar em uma linha separada.

Estilizando suas instruções if

Em todos os exemplos deste capítulo vimos bons hábitos de estilização. A única recomendação fornecida pela PEP 8 para estilizar testes condicionais é usar um único espaço em torno dos operadores de comparação, como `==`, `>=` e `<=`. Por exemplo: if age < 4: é melhor que: if age<4: Esse espaçamento não afeta o modo como Python interpreta seu código; ele simplesmente deixa seu código mais fácil de ler para você e para outras pessoas.

FAÇA VOCÊ MESMO

5.12 – Estilizando instruções `if`: Revise os programas que você escreveu neste capítulo e certifique-se de que os testes condicionais foram estilizados de forma apropriada.

5.13 – Suas ideias: A essa altura, você é um programador mais capacitado do que era quando começou a ler este livro. Agora que você tem melhor noção de como situações do mundo real são modeladas em programas, talvez esteja pensando em alguns problemas que poderia resolver com seus próprios programas. Registre qualquer ideia nova que tiver sobre problemas que você queira resolver à medida que suas habilidades em programação continuam a melhorar. Considere jogos que você queira escrever, conjuntos de dados que possa querer explorar e aplicações web que gostaria de criar.

Resumo

Neste capítulo aprendemos a escrever testes condicionais, que são sempre avaliados como `True` ou `False`. Vimos como escrever instruções `if` simples, cadeias `if-else` e cadeias `if-elif-else`. Começamos a usar essas estruturas para identificar condições particulares que deviam ser testadas e aprendemos a reconhecer quando essas condições foram atendidas em nossos programas. Aprendemos a tratar determinados itens de uma lista de modo diferente de todos os demais itens, ao mesmo tempo que continuamos usando a eficiência do laço `for`. Também retomamos as recomendações de estilo de Python para garantir que seus programas cada vez mais complexos continuem relativamente fáceis de ler e de entender.

No Capítulo 6 conheceremos os dicionários de Python. Um dicionário é semelhante a uma lista, mas permite conectar informações. Aprenderemos a criar dicionários, percorrê-los com laços e usá-los em conjunto com listas e instruções `if`. Conhecer os dicionários permitirá modelar uma variedade ainda maior de situações do mundo real.

7
Entrada de usuário e laços while

A maioria dos programas é escrita para resolver o problema de um usuário final. Para isso, geralmente precisamos obter algumas informações do usuário. Como exemplo simples, vamos supor que uma pessoa queira descobrir se ela tem idade suficiente para votar. Se você escrever um programa que responda a essa pergunta, será necessário saber a idade do usuário antes de poder oferecer uma resposta. O programa precisará pedir ao usuário que forneça a idade, ou *entre* com ela; depois que tiver esse dado de entrada, o programa poderá comparar esse valor com a idade para votar a fim de determinar se o usuário tem idade suficiente e então informar o resultado.

Neste capítulo aprenderemos a aceitar dados de entrada do usuário para que seu programa possa então trabalhar com eles. Quando seu programa precisar de um nome, você poderá solicitá-lo ao usuário. Quando precisar de uma lista de nomes, você poderá pedir uma série de nomes ao usuário. Para isso, usaremos a função `input()`.

Também veremos como manter os programas executando pelo tempo que os usuários quiserem, de modo que eles possam fornecer quantas informações forem necessárias; em seguida, seu programa poderá trabalhar com essas informações. Usaremos o laço `while` de Python para manter os programas executando enquanto determinadas condições permanecerem verdadeiras.

Com a capacidade de trabalhar com dados de entrada do usuário e controlar o tempo que seus programas executam, você poderá escrever programas totalmente interativos.

Como a função input() trabalha

A função `input()` faz uma pausa em seu programa e espera o usuário

fornecer um texto. Depois que Python recebe a entrada do usuário, esse dado é armazenado em uma variável para que você possa trabalhar com ele de forma conveniente.

Por exemplo, o programa a seguir pede que o usuário forneça um texto e, em seguida, exibe essa mensagem de volta ao usuário: parrot.py message = input("Tell me something, and I will repeat it back to you: ") print(message) A função `input()` aceita um argumento: o *prompt* – ou as instruções – que queremos exibir ao usuário para que eles saibam o que devem fazer. Nesse exemplo, quando Python executar a primeira linha, o usuário verá o prompt `Tell me something, and I will repeat it back to you:` . O programa espera enquanto o usuário fornece sua resposta e continua depois que ele tecla ENTER. A resposta é armazenada na variável `message`; depois disso, `print(message)` exibe a entrada de volta ao usuário: Tell me something, and I will repeat it back to you: **`Hello everyone!`**

```
Hello everyone!
```

NOTA O Sublime Text não executa programas que pedem uma entrada ao usuário. Você pode usar o Sublime Text para escrever programas que solicitem uma entrada, mas será necessário executar esses programas a partir de um terminal. Consulte a seção "Executando programas Python a partir de um terminal".

Escrevendo prompts claros

Sempre que usar a função `input()`, inclua um prompt claro, fácil de compreender, que informe o usuário exatamente que tipo de informação você procura. Qualquer frase que diga aos usuários o que eles devem fornecer será apropriada. Por exemplo: greeter.py name = input("Please enter your name: ") print("Hello, " + name + "!") Acrescente um espaço no final de seus prompts (depois dos dois-pontos no exemplo anterior) para separar o prompt da resposta do usuário e deixar claro em que lugar o usuário deve fornecer seu texto. Por exemplo: Please enter your name: **`Eric`** Hello, Eric!

Às vezes, você vai querer escrever um prompt que seja maior que uma linha. Por exemplo, talvez você queira explicar ao usuário por que está pedindo determinada entrada. Você pode armazenar seu prompt em uma variável e passá-la para a função `input()`. Isso permite criar seu prompt com várias linhas e escrever uma instrução `input()` clara.

greeter.py prompt = "If you tell us who you are, we can personalize the messages you see."

```
prompt += "\nWhat is your first name? "


name = input(prompt) print("\nHello, " + name + "!") Esse exemplo mostra
uma maneira de criar uma string multilinha. A primeira linha armazena a
parte inicial da mensagem na variável prompt. Na segunda linha, o
operador += acrescenta a nova string no final da string que estava
armazenada em prompt.
```

O prompt agora ocupa duas linhas, novamente, com um espaço após o ponto de interrogação por questões de clareza: If you tell us who you are, we can personalize the messages you see.

```
What is your first name? Eric
Hello, Eric!
```

Usando int() para aceitar entradas numéricas

Se usarmos a função `input()`, Python interpretará tudo que o usuário fornecer como uma string. Considere a sessão de interpretador a seguir, que pergunta a idade do usuário: >>> **`age = input("How old are you? ")`** How old are you? **`21`**

```
>>> age '21'
```

O usuário digita o número 21, mas quando pedimos o valor de `age`, Python devolve `'21'`, que é a representação em string do valor numérico fornecido. Sabemos que Python interpretou a entrada como uma string porque o número agora está entre aspas. Se tudo que você quiser fazer é exibir a entrada, isso funcionará bem. Entretanto, se tentar usar a entrada como um número, você obterá um erro: >>> **`age = input("How old are you? ")`** How old are you? **`21`**

```
❶ >>> age >= 18
  Traceback (most recent call last): File "<stdin>", line 1, in <module> ❷
  TypeError: unorderable types: str() >= int() Quando tentamos usar a
  entrada para fazer uma comparação numérica ❶, Python gera um erro, pois
  não é capaz de comparar uma string com um inteiro: a string '21'
  armazenada em age não pode ser comparada ao valor numérico 18 ❷.
```

Podemos resolver esse problema usando a função `int()`, que diz a Python para tratar a entrada como um valor numérico. A função `int()` converte a representação em string de um número em uma representação numérica, como vemos a seguir: >>> **`age = input("How old are you? ")`** How old are you? **`21`**

```
❶ >>> age = int(age) >>> age >= 18
  True
```

Nesse exemplo, quando fornecemos `21` no prompt, Python interpreta o número como uma string, mas o valor é então convertido para uma representação numérica por `int()` ❶. Agora Python pode executar o teste condicional: comparar `age` (que contém o valor numérico 21) com `18` para ver se `age` é maior ou igual a 18. Esse teste é avaliado como `True`.

Como podemos usar a função `int()` em um programa de verdade? Considere um programa que determina se as pessoas têm altura suficiente para andar em uma montanha-russa: rollercoaster.py height = input("How tall are you, in inches? ") height = int(height)

```
if height >= 36: print("\nYou're tall enough to ride!") else:
  print("\nYou'll be able to ride when you're a little older.") O
programa é capaz de comparar height com 36 porque height = int(height)
converte o valor de entrada em uma representação numérica antes de fazer
a comparação. Se o número fornecido for maior ou igual a 36, diremos ao
usuário que sua altura é suficiente: How tall are you, in inches? 71


You're tall enough to ride!
```

Quando usar uma entrada numérica para fazer cálculos e comparações, lembre-se de converter o valor da entrada em uma representação numérica antes.

Operador de módulo

Uma ferramenta útil para trabalhar com informações numéricas é o *operador de módulo* (`%`), que divide um número por outro e devolve o resto: >>> **4 % 3**

```
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

O operador de módulo não diz quantas vezes um número cabe em outro; ele simplesmente informa o resto.

Quando um número é divisível por outro, o resto é 0, portanto o

operador de módulo sempre devolve 0 nesse caso. Podemos usar esse fato para determinar se um número é par ou ímpar: even_or_odd.py number = input("Enter a number, and I'll tell you if it's even or odd: ") number = int(number)

```
if number % 2 == 0: print("\nThe number " + str(number) + " is even.")
else:
  print("\nThe number " + str(number) + " is odd.") Números pares são
sempre divisíveis por dois, portanto, se o módulo entre um número e dois
for zero (nesse caso, if number % 2 == 0), o número será par. Caso
contrário, será ímpar.

Enter a number, and I'll tell you if it's even or odd: 42


The number 42 is even.
```

Aceitando entradas em Python 2.7

Se você usa Python 2.7, utilize a função `raw_input()` quando pedir uma entrada ao usuário. Essa função interpreta todas as entradas como uma string, como faz `input()` em Python 3.

Python 2.7 também tem uma função `input()`, mas essa função interpreta a entrada do usuário como código Python e tenta executá-la. No melhor caso, você verá um erro informando que Python não é capaz de compreender a entrada; no pior caso, executará um código que não pretendia executar. Se estiver usando Python 2.7, utilize `raw_input()` no lugar de `input()`.

FAÇA VOCÊ MESMO

7.1 – Locação de automóveis: Escreva um programa que pergunte ao usuário qual tipo de carro ele gostaria de alugar. Mostre uma mensagem sobre esse carro, por exemplo, "Deixe-me ver se consigo um Subaru para você."

7.2 – Lugares em um restaurante: Escreva um programa que pergunte ao usuário quantas pessoas estão em seu grupo para jantar. Se a resposta for maior que oito, exiba uma mensagem dizendo que eles deverão esperar uma mesa. Caso contrário, informe que sua mesa está pronta.

7.3 – Múltiplos de dez: Peça um número ao usuário e, em seguida, informe se o número é múltiplo de dez ou não.

Introdução aos laços while

O laço `for` toma uma coleção de itens e executa um bloco de código

uma vez para cada item da coleção. Em comparação, o laço `while` executa durante o tempo em que, ou *enquanto*, uma determinada condição for verdadeira.

Laço while em ação

Podemos usar um laço `while` para contar uma série de números. Por exemplo, o laço `while` a seguir conta de 1 a 5: counting.py current_number = 1

```
while current_number <= 5: print(current_number) current_number += 1
```

Na primeira linha começamos a contar de 1 ao definir o valor de `current_number` com 1. O laço `while` é então configurado para continuar executando enquanto o valor de `current_number` for menor ou igual a 5. O código no laço exibe o valor `current_number` e então soma 1 a esse valor com `current_number += 1`. (O operador `+=` é um atalho para `current_number = current_number + 1`.) Python repete o laço enquanto a condição `current_number <= 5` for verdadeira. Como 1 é menor que 5, Python exibe `1` e então soma 1, fazendo o número atual ser igual a `2`. Como 2 é menor que 5, Python exibe `2` e soma 1 novamente, fazendo o número atual ser igual a `3`, e assim sucessivamente. Quando o valor de `current_number` for maior que 5, o laço para de executar e o programa termina: 1

```
2
3
4
5
```

Os programas que você usa no dia a dia provavelmente contêm laços `while`. Por exemplo, um jogo precisa de um laço `while` para continuar executando enquanto você quiser jogar, e pode parar de executar assim que você pedir para sair. Os programas não seriam divertidos se parassem de executar antes que lhes disséssemos para parar ou se continuassem executando mesmo depois que quiséssemos sair, portanto os laços `while` são bem úteis.

Deixando o usuário decidir quando quer sair

Podemos fazer o programa *parrot.py* executar enquanto o usuário quiser colocar a maior parte dele em um laço `while`. Definiremos um *valor de saída* e então deixaremos o programa executando enquanto o usuário

não tiver fornecido o valor de saída: parrot.py ❶ prompt = "\nTell me something, and I will repeat it back to you:"

```
  prompt += "\nEnter 'quit' to end the program. "


❷ message = ""
❸ while message != 'quit': message = input(prompt) print(message) Em ❶
definimos um prompt que informa quais são as duas opções ao usuário:
fornecer uma mensagem ou o valor de saída (nesse caso, é 'quit'). Em
seguida, preparamos uma variável message ❷ para armazenar o valor que o
usuário fornecer. Definimos message como uma string vazia, "", de modo que
Python tenha algo para conferir na primeira vez que alcançar a linha com
while. Na primeira vez que o programa executar e Python alcançar a
instrução while, ele deverá comparar o valor de message com 'quit', mas o
usuário ainda não forneceu nenhuma entrada. Se Python não tiver nada para
comparar, ele não será capaz de continuar executando o programa. Para
resolver esse problema, garantimos que message receba um valor inicial.
Embora seja apenas uma string vazia, ela fará sentido para Python e
permitirá que a comparação que faz o laço while funcionar seja feita. Esse
laço while ❸ executa enquanto o valor de message não for 'quit'.
```

Na primeira passagem pelo laço, `message` é apenas uma string vazia, portanto Python entra no laço. Em `message = input(prompt)`, Python exibe o prompt e espera o usuário fornecer uma entrada. O que quer que seja fornecido será armazenado em `message` e exibido; em seguida, Python avalia novamente a condição na instrução `while`. Desde que o usuário não tenha fornecido a palavra `'quit'`, o prompt será exibido novamente e Python esperará mais entradas. Quando o usuário finalmente digitar `'quit'`, Python para de executar o laço `while` e o programa termina: Tell me something, and I will repeat it back to you: Enter 'quit' to end the program. **`Hello everyone!`**

```
  Hello everyone!


  Tell me something, and I will repeat it back to you: Enter 'quit' to end
  the program. Hello again.
  Hello again.


  Tell me something, and I will repeat it back to you: Enter 'quit' to end
  the program. quit quit
```

Esse programa funciona bem, exceto que exibe a palavra `'quit'` como se fosse uma mensagem de verdade. Um teste `if` simples corrige esse problema: `prompt = "\nTell me something, and I will repeat it`

```
back to you:"
  prompt += "\nEnter 'quit' to end the program. "

  message = ""
  while message != 'quit': message = input(prompt)
    if message != 'quit': print(message) Agora o programa faz uma
  verificação rápida antes de mostrar a mensagem e só a exibirá se ela não
  for igual ao valor de saída: Tell me something, and I will repeat it back
  to you: Enter 'quit' to end the program. Hello everyone!
  Hello everyone!

  Tell me something, and I will repeat it back to you: Enter 'quit' to end
  the program. Hello again.
  Hello again.

  Tell me something, and I will repeat it back to you: Enter 'quit' to end
  the program. quit
```

Usando uma flag

No exemplo anterior, tínhamos um programa que executava determinadas tarefas enquanto uma dada condição era verdadeira. E como ficaria em programas mais complicados, em que muitos eventos diferentes poderiam fazer o programa parar de executar?

Por exemplo, em um jogo, vários eventos diferentes podem encerrá-lo. Quando o jogador ficar sem espaçonaves, seu tempo se esgotar ou as cidades que ele deveria proteger forem todas destruídas, o jogo deverá terminar. O jogo termina se qualquer um desses eventos ocorrer. Se muitos eventos possíveis puderem ocorrer para o programa terminar, tentar testar todas essas condições em uma única instrução `while` torna-se complicado e difícil.

Para um programa que deva executar somente enquanto muitas condições forem verdadeiras, podemos definir uma variável que determina se o programa como um todo deve estar ativo. Essa variável, chamada de *flag*, atua como um sinal para o programa. Podemos escrever nossos programas de modo que executem enquanto a flag estiver definida com `True` e parem de executar quando qualquer um dos vários eventos definir o valor da flag com `False`. Como resultado, nossa instrução `while` geral precisa verificar apenas uma condição: se a flag, no

momento, é `True`. Então todos os nossos demais testes (para ver se um evento que deve definir a flag com `False` ocorreu) podem estar bem organizados no restante do programa.

Vamos adicionar uma flag em *parrot.py*, que vimos na seção anterior. Essa flag, que chamaremos de `active` (embora você possa lhe dar o nome que quiser), controlará se o programa deve ou não continuar executando: `prompt = "\nTell me something, and I will repeat it back to you:"`

```
  prompt += "\nEnter 'quit' to end the program. "

❶ active = True ❷ while active: message = input(prompt)
❸ if message == 'quit': active = False ❹ else: print(message) Definimos a
variável active com True ❶ para que o programa comece em um estado ativo.
Fazer isso simplifica a instrução while, pois nenhuma comparação é feita
nessa instrução; a lógica é tratada em outras partes do programa. Enquanto
a variável active permanecer True, o laço continuará a executar ❷.
```

Na instrução `if` contida no laço `while`, verificamos o valor de `message` depois que o usuário fornece sua entrada. Se o usuário fornecer `'quit'` ❸, definimos `active` com `False` e o laço `while` é encerrado. Se o usuário fornecer outro dado que não seja `'quit'` ❹, exibimos essa entrada como uma mensagem.

Esse programa gera a mesma saída do exemplo anterior, em que havíamos colocado o teste condicional diretamente na instrução `while`. Porém, agora que temos uma flag para indicar se o programa como um todo está em um estado ativo, será mais fácil acrescentar outros testes (por exemplo, instruções `elif`) para eventos que devam fazer `active` se tornar `False`. Isso é útil em programas complicados, como jogos, em que pode haver muitos eventos, e qualquer um deles poderia fazer o programa parar de executar. Quando um desses eventos fizer a flag `active` se tornar `False`, o laço principal do jogo terminará, uma mensagem de *Game Over* poderia ser exibida e o jogador poderia ter a opção de jogar novamente.

Usando break para sair de um laço

Para sair de um laço `while` de imediato, sem executar qualquer código restante no laço, independentemente do resultado de qualquer teste condicional, utilize a instrução `break`. A instrução `break` direciona o fluxo

de seu programa; podemos usá-la para controlar quais linhas de código são ou não são executadas, de modo que o programa execute apenas o código que você quiser, quando você quiser.

Por exemplo, considere um programa que pergunte aos usuários os nomes de lugares que eles já visitaram. Podemos interromper o laço `while` nesse programa chamando `break` assim que o usuário fornecer o valor `'quit'`: cities.py prompt = "\nPlease enter the name of a city you have visited:"

```
prompt += "\n(Enter 'quit' when you are finished.) "


❶ while True: city = input(prompt)
    if city == 'quit': break else: print("I'd love to go to " +
  city.title() + "!") Um laço que comece com while True ❶ executará
  indefinidamente, a menos que alcance uma instrução break. O laço desse
  programa continuará pedindo aos usuários para que entrem com os nomes das
  cidades em que eles estiveram até que 'quit' seja fornecido. Quando
  'quit' for digitado, a instrução break é executada, fazendo Python sair
  do laço: Please enter the name of a city you have visited: (Enter 'quit'
  when you are finished.) New York I'd love to go to New York!


  Please enter the name of a city you have visited: (Enter 'quit' when you
  are finished.) San Francisco I'd love to go to San Francisco!


  Please enter the name of a city you have visited: (Enter 'quit' when you
  are finished.) quit NOTA Você pode usar a instrução break em qualquer
  laço de Python. Por exemplo, break pode ser usado para sair de um laço
  for que esteja percorrendo uma lista ou um dicionário.
```

Usando continue em um laço

Em vez de sair totalmente de um laço sem executar o restante de seu código, podemos usar a instrução `continue` para retornar ao início, com base no resultado de um teste condicional. Por exemplo, considere um laço que conte de 1 a 10, mas apresente apenas os números ímpares desse intervalo: counting.py current_number = 0

```
while current_number < 10: ❶ current_number += 1
  if current_number % 2 == 0: continue
```

```
  print(current_number) Inicialmente, definimos current_number com 0.
Como esse valor é menor que 10, Python entra no laço while. Uma vez no
laço, incrementamos o contador de 1 em ❶, portanto current_number passa
a ser 1. A instrução if então verifica o módulo entre current_number e 2.
Se o módulo for 0 (o que significa que current_number é divisível por 2),
a instrução continue diz a Python para ignorar o restante do laço e
voltar ao início. Se o número atual não for divisível por 2, o restante
do laço será executado e Python exibirá o número atual: 1

3
5
7
9
```

Evitando loops infinitos

Todo laço `while` precisa de uma maneira de interromper a execução para que não continue executando indefinidamente. Por exemplo, o laço a seguir deve fazer a contagem de 1 a 5: counting.py x = 1

```
while x <= 5: print(x) x += 1
```

Contudo, se você omitir a linha `x += 1` (como vemos a seguir) por acidente, o laço executará para sempre: # Esse laço executa indefinidamente!

```
x = 1
while x <= 5: print(x) Agora o valor de x começará em 1, mas jamais será
modificado. Como resultado, o teste condicional x <= 5 será sempre
avaliado como True e o laço while executará indefinidamente, exibindo uma
série de 1s, assim: 1
1
1
1
--trecho omitido--
```

Todo programador escreve ocasionalmente um loop infinito (ou laço infinito) com `while` por acidente, em especial quando os laços do programa tiverem condições de saída sutis. Se seu programa ficar preso em um loop infinito, tecle CTRL-C ou simplesmente feche a janela do terminal que está exibindo a saída de seu programa.

Para evitar escrever loops infinitos, teste todos os laços `while` e certifique-se de que eles serão encerrados conforme esperado. Se quiser que seu programa termine quando o usuário fornecer determinado valor de entrada, execute o programa e forneça esse valor. Se o programa não

terminar, analise cuidadosamente o modo como seu programa trata o valor que deveria fazer o laço parar. Garanta que pelo menos uma parte do programa possa fazer a condição do laço ser `False` ou fazer uma instrução `break` ser alcançada.

NOTA Alguns editores, como o Sublime Text, tem uma janela de saída incluída. Isso pode dificultar a interrupção de um loop infinito, e talvez seja necessário fechar o editor para encerrar o laço.

FAÇA VOCÊ MESMO

7.4 – Ingredientes para uma pizza: Escreva um laço que peça ao usuário para fornecer uma série de ingredientes para uma pizza até que o valor `'quit'` seja fornecido. À medida que cada ingrediente é especificado, apresente uma mensagem informando que você acrescentará esse ingrediente à pizza.

7.5 – Ingressos para o cinema: Um cinema cobra preços diferentes para os ingressos de acordo com a idade de uma pessoa. Se uma pessoa tiver menos de 3 anos de idade, o ingresso será gratuito; se tiver entre 3 e 12 anos, o ingresso custará 10 dólares; se tiver mais de 12 anos, o ingresso custará 15 dólares. Escreva um laço em que você pergunte a idade aos usuários e, então, informe-lhes o preço do ingresso do cinema.

7.6 – Três saídas: Escreva versões diferentes do Exercício *7.4* ou do Exercício *7.5* que faça o seguinte, pelo menos uma vez: • use um teste condicional na instrução `while` para encerrar o laço; • use uma variável `active` para controlar o tempo que o laço executará; • use uma instrução `break` para sair do laço quando o usuário fornecer o valor `'quit'`.

7.7 – Infinito: Escreva um laço que nunca termine e execute-o. (Para encerrar o laço, pressione CTRL-C ou feche a janela que está exibindo a saída.)

Usando um laço while com listas e dicionários

Até agora trabalhamos com apenas uma informação do usuário a cada vez. Recebemos a entrada do usuário e então a exibimos ou apresentamos uma resposta a ela. Na próxima passagem pelo laço `while`, recebíamos outro valor de entrada e respondíamos a ela. Contudo, para controlar muitos usuários e informações, precisaremos usar listas e dicionários com nossos laços `while`.

Um laço `for` é eficiente para percorrer uma lista, mas você não deve modificar uma lista em um laço `for`, pois Python terá problemas para manter o controle dos itens da lista. Para modificar uma lista enquanto trabalhar com ela, utilize um laço `while`. Usar laços `while` com listas e dicionários permite coletar, armazenar e organizar muitas entradas a fim

de analisá-las e apresentá-las posteriormente.

Transferindo itens de uma lista para outra

Considere uma lista de usuários recém-registrados em um site, porém não verificados. Depois de conferir esses usuários, como podemos transferi-los para uma lista separada de usuários confirmados? Uma maneira seria usar um laço `while` para extrair os usuários da lista de usuários não confirmados à medida que os verificarmos e então adicioná-los em uma lista separada de usuários confirmados. Esse código pode ter o seguinte aspecto: confirmed_users.py # Começa com os usuários que precisam ser verificados, # e com uma lista vazia para armazenar os usuários confirmados ❶ unconfirmed_users = ['alice', 'brian', 'candace']

```
confirmed_users = []


# Verifica cada usuário até que não haja mais usuários não confirmados #
Transfere cada usuário verificado para a lista de usuários confirmados ❷
while unconfirmed_users: ❸ current_user = unconfirmed_users.pop()
  print("Verifying user: " + current_user.title()) ❹
confirmed_users.append(current_user)
# Exibe todos os usuários confirmados print("\nThe following users have
been confirmed:") for confirmed_user in confirmed_users:
print(confirmed_user.title()) Começamos com uma lista de usuários não
confirmados em ❶ (Alice, Brian e Candace) e uma lista vazia para
armazenar usuários confirmados. O laço while em ❷ executa enquanto a
lista unconfirmed_users não estiver vazia. Nesse laço, a função pop() em
❸ remove os usuários não verificados, um de cada vez, do final de
unconfirmed_users. Nesse caso, como Candace é o último elemento da lista
unconfirmed_users, seu nome será o primeiro a ser removido, armazenado em
current_user e adicionado à lista confirmed_users em ❹. O próximo é
Brian e, depois, Alice.
```

Simulamos a confirmação de cada usuário exibindo uma mensagem de verificação e então adicionando-os à lista de usuários confirmados. À medida que a lista de usuários não confirmados diminui, a lista de usuários confirmados aumenta. Quando a lista de usuários não confirmados estiver vazia, o laço para e a lista de usuários confirmados é exibida: Verifying user: Candace Verifying user: Brian Verifying user: Alice

```
The following users have been confirmed: Candace
Brian
```

```
Alice
```

Removendo todas as instâncias de valores específicos de uma lista

No Capítulo 3 usamos `remove()` para remover um valor específico de uma lista. A função `remove()` era apropriada porque o valor em que estávamos interessados aparecia apenas uma vez na lista. Porém, e se quiséssemos remover da lista todas as instâncias de um valor?

Suponha que tenhamos uma lista de animais de estimação com o valor `'cat'` repetido várias vezes. Para remover todas as instâncias desse valor, podemos executar um laço `while` até `'cat'` não estar mais na lista, como vemos aqui: pets.py pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']

```
print(pets)


while 'cat' in pets: pets.remove('cat')
print(pets)
```

Começamos com uma lista contendo várias instâncias de `'cat'`. Após exibir a lista, Python entra no laço `while`, pois encontra o valor `'cat'` na lista pelo menos uma vez. Depois que entrar no laço, Python remove a primeira instância de `'cat'`, retorna à linha `while` e então entra novamente no laço quando descobre que `'cat'` ainda está na lista. Cada instância de `'cat'` é removida até que esse valor não esteja mais na lista; nesse momento, Python sai do laço e exibe a lista novamente: ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']

```
['dog', 'dog', 'goldfish', 'rabbit']
```

Preenchendo um dicionário com dados de entrada do usuário

Podemos pedir a quantidade de entrada que for necessária a cada passagem por um laço `while`. Vamos criar um programa de enquete em que cada passagem pelo laço solicita o nome do participante e uma resposta. Armazenaremos os dados coletados em um dicionário, pois queremos associar cada resposta a um usuário em particular: mountain_poll.py responses = {}

```
# Define uma flag para indicar que a enquete está ativa polling_active =
True
```

```
while polling_active: # Pede o nome da pessoa e a resposta ❶ name =
input("\nWhat is your name? ") response = input("Which mountain would you
like to climb someday? ")
  # Armazena a resposta no dicionário ❷ responses[name] = response
  # Descobre se outra pessoa vai responder à enquete ❸ repeat =
input("Would you like to let another person respond? (yes/ no) ") if
repeat == 'no': polling_active = False
# A enquete foi concluída. Mostra os resultados print("\n--- Poll Results
---") ❹ for name, response in responses.items(): print(name + " would
like to climb " + response + ".") O programa inicialmente define um
dicionário vazio (responses) e cria uma flag (polling_active) para
indicar que a enquete está ativa. Enquanto polling_active for True,
Python executará o código que está no laço while.
```

Nesse laço é solicitado ao usuário que entre com seu nome e uma montanha que gostaria de escalar ❶. Essa informação é armazenada no dicionário `responses` ❷, e uma pergunta é feita ao usuário para saber se ele quer que a enquete continue ❸. Se o usuário responder `yes`, o programa entrará no laço `while` novamente. Se responder `no`, a flag `polling_active` será definida com `False`, o laço `while` para de executar e o último bloco de código em ❹ exibe o resultado da enquete.

Se executar esse programa e fornecer exemplos de respostas, você deverá ver uma saída como esta: What is your name? **`Eric`** Which mountain would you like to climb someday? **`Denali`** Would you like to let another person respond? (yes/ no) **`yes`**

```
What is your name? Lynn Which mountain would you like to climb someday?
Devil's Thumb Would you like to let another person respond? (yes/ no) no
--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

FAÇA VOCÊ MESMO

7.8 – Lanchonete: Crie uma lista chamada `sandwich_orders` e a preencha com os nomes de vários sanduíches. Em seguida, crie uma lista vazia chamada `finished_sandwiches`. Percorra a lista de pedidos de sanduíches com um laço e mostre uma mensagem para cada pedido, por exemplo, `Preparei seu sanduíche de atum`. À medida que cada sanduíche for preparado, transfira-o para a lista de sanduíches prontos. Depois que todos os sanduíches estiverem prontos, mostre uma mensagem que liste cada sanduíche preparado.

7.9 – Sem pastrami: Usando a lista `sandwich_orders` do Exercício 7.8, garanta que o sanduíche de `'pastrami'` apareça na lista pelo menos três vezes. Acrescente um código próximo ao início de seu programa para exibir uma

mensagem informando que a lanchonete está sem pastrami e, então, use um laço `while` para remover todas as ocorrências de `'pastrami'` e `sandwich_orders`. Garanta que nenhum sanduíche de pastrami acabe em `finished_sandwiches`.

7.10 – Férias dos sonhos: Escreva um programa que faça uma enquete sobre as férias dos sonhos dos usuários. Escreva um prompt semelhante a este: *Se pudesse visitar um lugar do mundo, para onde você iria?* Inclua um bloco de código que apresente os resultados da enquete.

Resumo

Neste capítulo aprendemos a usar `input()` para permitir que os usuários forneçam suas próprias informações em seus programas. Vimos como trabalhar com entradas tanto textuais quanto numéricas e a usar laços `while` para deixar seus programas executarem pelo tempo que os usuários quiserem. Conhecemos várias maneiras de controlar o fluxo de um laço `while`: definindo uma flag `active`, usando a instrução `break` e com a instrução `continue`. Aprendemos a usar um laço `while` para transferir itens de uma lista para outra e vimos como remover todas as instâncias de um valor de uma lista. Também vimos como laços `while` podem ser usados com dicionários.

No Capítulo 8, conheceremos as *funções*. As funções permitem dividir seus programas em partes menores, cada uma fazendo uma tarefa específica. Podemos chamar uma função quantas vezes quisermos e armazená-las em arquivos separados. Ao usar funções, podemos escrever códigos mais eficientes, em que seja mais fácil resolver problemas e dar manutenção, com possibilidade de serem reutilizados em vários programas diferentes.