8
Funções

Neste capítulo aprenderemos a escrever *funções*, que são blocos de código nomeados, concebidos para realizar uma tarefa específica. Quando queremos executar uma tarefa em particular, definida em uma função, *chamamos* o nome da função responsável por ela. Se precisar executar essa tarefa várias vezes durante seu programa, não será necessário digitar todo o código para a mesma tarefa repetidamente: basta chamar a função dedicada ao tratamento dessa tarefa e a chamada dirá a Python para executar o código da função. Você perceberá que usar funções permite escrever, ler, testar e corrigir seus programas de modo mais fácil.

Neste capítulo também veremos maneiras de passar informações às funções. Aprenderemos a escrever determinadas funções cuja tarefa principal seja exibir informações e outras funções que visam a processar dados e devolver um valor ou um conjunto de valores. Por fim, veremos como armazenar funções em arquivos separados, chamados de *módulos*, para ajudar a organizar os arquivos principais de seu programa.

Definindo uma função

Eis uma função simples chamada `greet_user()` que exibe uma saudação: greeter.py ❶ def greet_user(): ❷ """Exibe uma saudação simples."""
❸ `print("Hello!")`
❹ `greet_user()` Esse exemplo mostra a estrutura mais simples possível para uma função. A linha em ❶ utiliza a palavra reservada def para informar Python que estamos definindo uma função. Essa é a *definição da função*, que informa o nome da função a Python e, se for aplicável, quais são os tipos de informação necessários à função para que ela faça sua tarefa. Os parênteses contêm essa informação. Nesse caso, o nome da função é `greet_user()`, e ela não precisa de nenhuma informação para executar sua tarefa, portanto os parênteses estão vazios. (Mesmo assim, eles são obrigatórios.) Por fim, a definição termina com dois-pontos.

Qualquer linha indentada após `def greet_user():` faz parte do *corpo* da função. O texto em ❷ é um comentário chamado *docstring*, que descreve o que a função faz. As docstrings são colocadas entre aspas triplas, que Python procura quando gera a documentação das funções de seus programas.

A linha `print("Hello!")` ❸ é a única linha com código propriamente dito no corpo dessa função, portanto `greet_user()` realiza apenas uma tarefa: `print("Hello!")`.

Quando quiser usar essa função, você deve chamá-la. Uma *chamada de função* diz a Python para executar o código da função. Para *chamar* uma função, escreva o nome dela, seguido de qualquer informação necessária entre parênteses, como vemos em ❹. Como nenhuma informação é necessária nesse caso, chamar nossa função é simples e basta fornecer `greet_user()`. Como esperado, ela exibe `Hello!`: Hello!

Passando informações para uma função

Se for um pouco modificada, a função `greet_user()` não só pode dizer `Hello!` ao usuário como também pode saudá-lo pelo nome. Para que a função faça isso, especifique `username` entre os parênteses da definição da função em `def greet_user()`. Ao acrescentar `username` aqui, permitimos que a função aceite qualquer valor que você especificar para `username`. A função agora espera que um valor seja fornecido para `username` sempre que ela for chamada. Ao chamar `greet_user()`, você poderá lhe passar um nome, por exemplo, `'jesse'`, entre parênteses: def greet_user(username): """Exibe uma saudação simples."""

```
    print("Hello, " + username.title() + "!")
greet_user('jesse') Usar greet_user('jesse') faz greet_user() ser chamada
e fornece as informações de que a função precisa para executar a
instrução print. A função aceita o nome que você passar e exibe a
saudação para esse nome: Hello, Jesse!
```

De modo semelhante, usar `greet_user('sarah')` chama `greet_user()`, passa `'sarah'` a essa função e exibe `Hello, Sarah!`. Você pode chamar `greet_user()` quantas vezes quiser e lhe passar qualquer nome desejado de modo a gerar sempre uma saída previsível.

Argumentos e parâmetros

Na função `greet_user()` anterior, definimos `greet_user()` para que exija um valor para a variável `username`. Depois que chamamos a função e lhe fornecemos a informação (o nome de uma pessoa), a saudação correta foi exibida.

A variável `username` na definição de `greet_user()` é um exemplo de *parâmetro* – uma informação de que a função precisa para executar sua tarefa. O valor `'jesse'` em `greet_user('jesse')` é um exemplo de *argumento*. Um argumento é uma informação passada para uma função em sua chamada. Quando chamamos a função, colocamos entre parênteses o valor com que queremos que a função trabalhe. Nesse caso, o argumento `'jesse'` foi passado para a função `greet_user()` e o valor foi armazenado no parâmetro `username`.

NOTA Às vezes, as pessoas falam de argumentos e parâmetros de modo indistinto. Não fique surpreso se vir as variáveis de uma definição de função serem referenciadas como argumentos, ou as variáveis de uma chamada de função serem chamadas de parâmetros.

FAÇA VOCÊ MESMO

8.1 – Mensagem: Escreva uma função chamada `display_message()` que mostre uma frase informando a todos o que você está aprendendo neste capítulo. Chame a função e certifique-se de que a mensagem seja exibida corretamente.

8.2 – Livro favorito: Escreva uma função chamada `favorite_book()` que aceite um parâmetro `title`. A função deve exibir uma mensagem como `Um dos meus livros favoritos é Alice no país das maravilhas.` Chame a função e não se esqueça de incluir o título do livro como argumento na chamada da função.

Passando argumentos

Pelo fato de ser possível que uma definição de função tenha vários parâmetros, uma chamada de função pode precisar de diversos argumentos. Os argumentos podem ser passados para as funções de várias maneiras. Podemos usar *argumentos posicionais*, que devem estar na mesma ordem em que os parâmetros foram escritos, *argumentos nomeados* (keyword arguments), em que cada argumento é constituído de um nome de variável e de um valor, ou por meio de listas e dicionários de valores. Vamos analisar cada um deles.

Argumentos posicionais

Quando chamamos uma função, Python precisa fazer a correspondência entre cada argumento da chamada da função e um parâmetro da definição. A maneira mais simples de fazer isso é contar com a ordem dos argumentos fornecidos. Valores cuja correspondência seja feita dessa maneira são chamados de *argumentos posicionais*.

Para ver como isso funciona considere uma função que apresente informações sobre animais de estimação. A função nos informa o tipo de cada animal de estimação e o nome dele, como vemos aqui: pets.py ❶ def describe_pet(animal_type, pet_name): """Exibe informações sobre um animal de estimação."""

```
    print("\nI have a " + animal_type + ".") print("My " + animal_type +
  "'s name is " + pet_name.title() + ".")
```

❷ describe_pet('hamster', 'harry') A definição mostra que essa função precisa de um tipo de animal e de seu nome ❶. Quando chamamos describe_pet(), devemos fornecer o tipo do animal e um nome, nessa ordem. Por exemplo, na chamada da função, o argumento 'hamster' é armazenado no parâmetro animal_type e o argumento 'harry' é armazenado no parâmetro pet_name ❷. No corpo da função, esses dois parâmetros são usados para exibir informações sobre o animal de estimação descrito.

A saída apresenta um hamster chamado Harry: I have a hamster.

```
My hamster's name is Harry.
```

Várias chamadas de função

Podemos chamar uma função quantas vezes forem necessárias. Descrever um segundo animal de estimação diferente exige apenas mais uma chamada a describe_pet(): def describe_pet(animal_type, pet_name): """Exibe informações sobre um animal de estimação."""

```
    print("\nI have a " + animal_type + ".") print("My " + animal_type +
  "'s name is " + pet_name.title() + ".")
  describe_pet('hamster', 'harry') describe_pet('dog', 'willie') Nessa
  segunda chamada da função, passamos os argumentos 'dog' e 'willie' a
  describe_pet(). Assim como no conjunto anterior de argumentos que usamos,
  Python faz a correspondência entre 'dog' e o parâmetro animal_type e
  entre 'willie' e o parâmetro pet_name. Como antes, a função faz sua
  tarefa, porém, dessa vez, exibe valores para um cachorro chamado Willie.
  Agora temos um hamster chamado Harry e um cachorro chamado Willie: I have
  a hamster.
  My hamster's name is Harry.


  I have a dog.
```

```
My dog's name is Willie.
```

Chamar uma função várias vezes é uma maneira eficiente de trabalhar. O código que descreve um animal de estimação é escrito uma só vez na função. Então, sempre que quiser descrever um novo animal de estimação, podemos chamar a função com as informações sobre esse animal. Mesmo que o código para descrever um animal de estimação fosse expandido atingindo dez linhas, poderíamos ainda descrever um novo animal de estimação chamando a função novamente com apenas uma linha.

Podemos usar tantos argumentos posicionais quantos forem necessários nas funções. Python trabalha com os argumentos fornecidos na chamada da função e faz a correspondência de cada um com o parâmetro associado na definição da função.

A ordem é importante em argumentos posicionais

Podemos obter resultados inesperados se confundirmos a ordem dos argumentos em uma chamada de função quando argumentos posicionais forem usados: `def describe_pet(animal_type, pet_name): """`Exibe informações sobre um animal de estimação.`"""`

```
    print("\nI have a " + animal_type + ".") print("My " + animal_type +
"'s name is " + pet_name.title() + ".")
describe_pet('harry', 'hamster') Nessa chamada de função, listamos
primeiro o nome e depois o tipo do animal. Como dessa vez o argumento
'harry' foi definido antes, esse valor é armazenado no parâmetro
animal_type. De modo semelhante, 'hamster' é armazenado em pet_name.
Agora temos um “harry” chamado “Hamster”: I have a harry.

My harry's name is Hamster.
```

Se obtiver resultados engraçados como esse, verifique se a ordem dos argumentos em sua chamada de função corresponde à ordem dos parâmetros na definição da função.

Argumentos nomeados

Um *argumento nomeado* (keyword argument) é um par nome-valor passado para uma função. Associamos diretamente o nome e o valor no próprio argumento para que não haja confusão quando ele for passado para a função (você não acabará com um harry chamado Hamster). Argumentos nomeados fazem com que você não precise se preocupar

com a ordem correta de seus argumentos na chamada da função e deixam claro o papel de cada valor na chamada.

Vamos reescrever *pets.py* usando argumentos nomeados para chamar describe_pet(): def describe_pet(animal_type, pet_name): """Exibe informações sobre um animal de estimação."""
print("\nI have a " + animal_type + ".") print("My " + animal_type + "'s name is " + pet_name.title() + ".")
describe_pet(animal_type='hamster', pet_name='harry') A função describe_pet() não mudou. Entretanto, quando chamamos a função, dizemos explicitamente a Python a qual parâmetro cada argumento deve corresponder. Quando Python lê a chamada da função, ele sabe que deve armazenar o argumento 'hamster' no parâmetro animal_type e o argumento 'harry' em pet_name. A saída mostra corretamente que temos um hamster chamado Harry.

A ordem dos argumentos nomeados não importa, pois Python sabe o que é cada valor. As duas chamadas de função a seguir são equivalentes: describe_pet(animal_type='hamster', pet_name='harry') describe_pet(pet_name='harry', animal_type='hamster') **NOTA** Quando usar argumentos nomeados, lembre-se de usar os nomes exatos dos parâmetros usados na definição da função.

Valores default

Ao escrever uma função, podemos definir um *valor default* para cada parâmetro. Se um argumento para um parâmetro for especificado na chamada da função, Python usará o valor desse argumento. Se não for, o valor default do parâmetro será utilizado. Portanto, se um valor default for definido para um parâmetro, você poderá excluir o argumento correspondente, que normalmente seria especificado na chamada da função. Usar valores default pode simplificar suas chamadas de função e deixar mais claro o modo como suas funções normalmente são utilizadas.

Por exemplo, se perceber que a maioria das chamadas a describe_pet() é usada para descrever cachorros, você pode definir o valor default de animal_type com 'dog'. Agora qualquer pessoa que chamar describe_pet() para um cachorro poderá omitir essa informação: def describe_pet(pet_name, animal_type='dog'): """Exibe informações sobre um animal de estimação."""

```
  print("\nI have a " + animal_type + ".") print("My " + animal_type +
"'s name is " + pet_name.title() + ".")
describe_pet(pet_name='willie') Mudamos a definição de describe_pet()
para incluir um valor default igual a 'dog' para animal_type. A partir de
agora, quando a função for chamada sem um animal_type especificado,
Python saberá que deve usar o valor 'dog' para esse parâmetro: I have a
dog.

My dog's name is Willie.
```

Observe que a ordem dos parâmetros na definição da função precisou ser alterada. Como o uso do valor default faz com que não seja necessário especificar um tipo de animal como argumento, o único argumento restante na chamada da função é o nome do animal de estimação. Python continua interpretando esse valor como um argumento posicional, portanto, se a função for chamada somente com o nome de um animal de estimação, esse argumento corresponderá ao primeiro parâmetro listado na definição da função. Esse é o motivo pelo qual o primeiro parâmetro deve ser `pet_name`.

O modo mais simples de usar essa função agora é fornecer apenas o nome de um cachorro na chamada da função: describe_pet('willie') Essa chamada de função produzirá a mesma saída do exemplo anterior. O único argumento fornecido é `'willie'`, portanto ele é associado ao primeiro parâmetro da definição da função, que é `pet_name`. Como nenhum argumento foi fornecido para `animal_type`, Python usa o valor default, que é `'dog'`.

Para descrever um animal que não seja um cachorro, uma chamada de função como esta pode ser usada: describe_pet(pet_name='harry', animal_type='hamster') Como um argumento explícito para `animal_type` foi especificado, Python ignorará o valor default do parâmetro.

NOTA Ao usar valores default, qualquer parâmetro com um valor desse tipo deverá ser listado após todos os parâmetros que não tenham valores default. Isso permite que Python continue a interpretar os argumentos posicionais corretamente.

Chamadas de função equivalentes

Como os argumentos posicionais, os argumentos nomeados e os valores default podem ser usados em conjunto, e com frequência você terá várias maneiras equivalentes de chamar uma função. Considere a

definição a seguir de `describe_pets()` com um valor default especificado: def describe_pet(pet_name, animal_type='dog'): Com essa definição, um argumento sempre deverá ser fornecido para `pet_name` e esse valor pode ser especificado por meio do formato posicional ou nomeado. Se o animal descrito não for um cachorro, um argumento para `animal_type` deverá ser incluído na chamada, e esse argumento também pode ser especificado com o formato posicional ou nomeado.

Todas as chamadas a seguir serão adequadas a essa função: # Um cachorro chamado Willie describe_pet('willie') describe_pet(pet_name='willie')

```
# Um hamster chamado Harry describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry') Cada uma dessas
chamadas de função produzirá a mesma saída dos exemplos anteriores.
```

NOTA O estilo de chamada que você usar realmente não importa. Desde que suas chamadas de função gerem a saída desejada, basta usar o estilo que achar mais fácil de entender.

Evitando erros em argumentos

Quando começar a usar funções, não se surpreenda se você se deparar com erros sobre argumentos sem correspondência. Argumentos sem correspondência ocorrem quando fornecemos menos ou mais argumentos necessários à função para que ela realize sua tarefa. Por exemplo, eis o que acontece se tentarmos chamar `describe_pet()` sem argumentos: `def describe_pet(animal_type, pet_name): """`Exibe informações sobre um animal de estimação.`"""`

```
  print("\nI have a " + animal_type + ".") print("My " + animal_type +
"'s name is " + pet_name.title() + ".")
describe_pet()
```

Python reconhece que algumas informações estão faltando na chamada da função e o traceback nos informa quais são: Traceback (most recent call last): ❶ File "pets.py", line 6, in <module> ❷ describe_pet() ❸ TypeError: describe_pet() missing 2 required positional arguments: 'animal_

```
type' and 'pet_name'
```

Em ❶ o traceback nos informa em que local está o problema, o que nos permite olhar o código novamente e ver que algo deu errado em

nossa chamada de função. Em ❷ a chamada de função causadora do problema é apresentada. Em ❸ o traceback nos informa que há dois argumentos ausentes na chamada da função e mostra o nome desses argumentos. Se essa função estivesse em um arquivo separado, provavelmente poderíamos reescrever a chamada de forma correta sem precisar abrir esse arquivo e ler o código da função.

Python nos ajuda lendo o código da função e informando os nomes dos argumentos que devemos fornecer. Esse é outro motivo para dar nomes descritivos às suas variáveis e funções. Se fizer isso, as mensagens de erro de Python serão mais úteis para você e para qualquer pessoa que possa usar o seu código.

Se você fornecer argumentos demais, deverá obter um traceback semelhante, que poderá ajudá-lo a fazer uma correspondência correta entre a chamada da função e sua definição.

FAÇA VOCÊ MESMO

8.3 – Camiseta: Escreva uma função chamada `make_shirt()` que aceite um tamanho e o texto de uma mensagem que deverá ser estampada na camiseta. A função deve exibir uma frase que mostre o tamanho da camiseta e a mensagem estampada.

Chame a função uma vez usando argumentos posicionais para criar uma camiseta. Chame a função uma segunda vez usando argumentos nomeados.

8.4 – Camisetas grandes: Modifique a função `make_shirt()` de modo que as camisetas sejam grandes por default, com uma mensagem *Eu amo Python*. Crie uma camiseta grande e outra média com a mensagem default, e uma camiseta de qualquer tamanho com uma mensagem diferente.

8.5 – Cidades: Escreva uma função chamada `describe_city()` que aceite o nome de uma cidade e seu país. A função deve exibir uma frase simples, como `Reykjavik está localizada na Islândia`. Forneça um valor default ao parâmetro que representa o país. Chame sua função para três cidades diferentes em que pelo menos uma delas não esteja no país default.

Valores de retorno

Uma função nem sempre precisa exibir sua saída diretamente. Em vez disso, ela pode processar alguns dados e então devolver um valor ou um conjunto de valores. O valor devolvido pela função é chamado de *valor de retorno*. A instrução `return` toma um valor que está em uma função e o envia de volta à linha que a chamou. Valores de retorno permitem passar

boa parte do trabalho pesado de um programa para funções, o que pode simplificar o corpo de seu programa.

Devolvendo um valor simples

Vamos observar uma função que aceite um primeiro nome e um sobrenome e devolva um nome completo formatado de modo elegante: formatted_name.py ❶ def get_formatted_name(first_name, last_name): """Devolve um nome completo formatado de modo elegante."""
❷ full_name = first_name + ' ' + last_name ❸ return full_name.title()
❹ musician = get_formatted_name('jimi', 'hendrix') print(musician)

A definição de `get_formatted_name()` aceita um primeiro nome e um sobrenome ❶ como parâmetros. A função combina esses dois nomes, acrescenta um espaço entre eles e armazena o resultado em `full_name` ❷. O valor de `full_name` é convertido para que tenha letras iniciais maiúsculas e é devolvido para a linha que fez a chamada em ❸.

Quando chamamos uma função que devolve um valor, precisamos fornecer uma variável em que o valor de retorno possa ser armazenado. Nesse caso, o valor devolvido é armazenado na variável `musician` em ❹. A saída mostra um nome formatado de modo elegante, composto das partes do nome de uma pessoa: Jimi Hendrix

Pode parecer bastante trabalho para obter um nome formatado de forma elegante quando poderíamos simplesmente ter escrito: print("Jimi Hendrix") No entanto, quando consideramos trabalhar com um programa de grande porte, que precise armazenar muitos primeiros nomes e sobrenomes separadamente, funções como `get_formatted_name()` tornam-se muito convenientes. Armazenamos os primeiros nomes e os sobrenomes de forma separada e então chamamos essa função sempre que quisermos exibir um nome completo.

Deixando um argumento opcional

Às vezes, faz sentido criar um argumento opcional para que as pessoas que usarem a função possam optar por fornecer informações extras somente se quiserem. Valores default podem ser usados para deixar um argumento opcional.

Por exemplo, suponha que queremos expandir `get_formatted_name()` para que trate nomes do meio também. Uma primeira tentativa para

incluir nomes do meio poderia ter o seguinte aspecto: def get_formatted_name(first_name, middle_name, last_name): """Devolve um nome completo formatado de modo elegante."""

```
    full_name = first_name + ' ' + middle_name + ' ' + last_name return
  full_name.title()
  musician = get_formatted_name('john', 'lee', 'hooker') print(musician)
```

Essa função é apropriada quando fornecemos um primeiro nome, um nome do meio e um sobrenome. Ela aceita todas as três partes de um nome e então compõe uma string a partir delas. A função acrescenta espaços nos pontos apropriados e converte o nome completo para que as iniciais sejam maiúsculas: John Lee Hooker No entanto, nomes do meio nem sempre são necessários, e essa função, conforme está escrita, não seria apropriada se tentássemos chamá-la somente com um primeiro nome e um sobrenome. Para deixar o nome do meio opcional, podemos associar um valor default vazio ao argumento `middle_name` e ignorá-lo, a menos que o usuário forneça um valor. Para que `get_formatted_name()` funcione sem um nome do meio, definimos o valor default de `middle_name` com uma string vazia e o passamos para o final da lista de parâmetros: ❶ def get_formatted_name(first_name, last_name, middle_name=''): """Devolve um nome completo formatado de modo elegante."""

```
❷ if middle_name: full_name = first_name + ' ' + middle_name + ' ' +
last_name ❸ else: full_name = first_name + ' ' + last_name return
full_name.title()
  musician = get_formatted_name('jimi', 'hendrix') print(musician)


❹ musician = get_formatted_name('john', 'hooker', 'lee') print(musician)
```

Nesse exemplo, o nome é criado a partir de três partes possíveis. Como o primeiro nome e o sobrenome sempre existem, esses parâmetros são listados antes na definição da função. O nome do meio é opcional, portanto é listado por último na definição, e seu valor default é uma string vazia ❶.

No corpo da função verificamos se um nome do meio foi especificado. Python interpreta strings não vazias como `True`, portanto `if middle_name` será avaliado como `True` se um argumento para o nome do meio estiver na chamada da função ❷. Se um nome do meio for especificado, o primeiro nome, o nome do meio e o sobrenome serão combinados para compor um nome completo. Esse nome é então alterado para que as iniciais sejam maiúsculas e é devolvido para a linha que chamou a

função: ele será armazenado em uma variável `musician` e exibido. Se um nome do meio não for especificado, a string vazia falhará no teste `if` e o bloco `else` será executado ❸. O nome completo será composto apenas do primeiro nome e do sobrenome, e o nome formatado é devolvido para a linha que fez a chamada: ele será armazenado em `musician` e exibido.

Chamar essa função com um primeiro nome e um sobrenome é simples. Se usarmos um nome do meio, porém, precisamos garantir que esse nome seja o último argumento passado para que Python faça a correspondência dos argumentos posicionais de forma correta ❹.

Essa versão modificada de nossa função é apropriada para pessoas que tenham apenas um primeiro nome e um sobrenome, mas funciona também para pessoas que tenham um nome do meio: Jimi Hendrix

```
John Lee Hooker
```

Valores opcionais permitem que as funções tratem uma grande variedade de casos de uso, ao mesmo tempo que simplificam ao máximo as chamadas de função.

Devolvendo um dicionário

Uma função pode devolver qualquer tipo de valor necessário, incluindo estruturas de dados mais complexas como listas e dicionários. Por exemplo, a função a seguir aceita partes de um nome e devolve um dicionário que representa uma pessoa: person.py def build_person(first_name, last_name): """Devolve um dicionário com informações sobre uma pessoa."""

```
❶ person = {'first': first_name, 'last': last_name}
❷ return person
  musician = build_person('jimi', 'hendrix') ❸ print(musician) A função
  build_person() aceita um primeiro nome e um sobrenome e então reúne esses
  valores em um dicionário em ❶. O valor de first_name é armazenado com a
  chave 'first' e o valor de last_name é armazenado com a chave 'last'. O
  dicionário completo que representa a pessoa é devolvido em ❷. O valor de
  retorno é exibido em ❸, com as duas informações textuais originais agora
  armazenadas em um dicionário: {'first': 'jimi', 'last': 'hendrix'}
```

Essa função aceita informações textuais simples e as coloca em uma estrutura de dados mais significativa, que permite trabalhar com as informações além de simplesmente exibi-las. As strings `'jimi'` e `'hendrix'` agora estão identificadas como um primeiro nome e um sobrenome.

Podemos facilmente estender essa função para que aceite valores opcionais como um nome do meio, uma idade, uma profissão ou qualquer outra informação que você queira armazenar sobre uma pessoa. Por exemplo, a alteração a seguir permite armazenar a idade de uma pessoa também: def build_person(first_name, last_name, age=''): """Devolve um dicionário com informações sobre uma pessoa."""
person = {'first': first_name, 'last': last_name}
if age: person['age'] = age return person
musician = build_person('jimi', 'hendrix', age=27) print(musician)
Adicionamos um novo parâmetro opcional age à definição da função e atribuímos um valor default vazio ao parâmetro. Se a chamada da função incluir um valor para esse parâmetro, ele será armazenado no dicionário. Essa função sempre armazena o nome de uma pessoa, mas também pode ser modificada para guardar outras informações que você quiser sobre ela.

Usando uma função com um laço while

Podemos usar funções com todas as estruturas Python que conhecemos até agora. Por exemplo, vamos usar a função `get_formatted_name()` com um laço `while` para saudar os usuários de modo mais formal. Eis uma primeira tentativa de saudar pessoas usando seu primeiro nome e o sobrenome: greeter.py `def get_formatted_name(first_name, last_name):` """Devolve um nome completo formatado de modo elegante."""
full_name = first_name + ' ' + last_name return full_name.title()
# Este é um loop infinito!
while True:
❶ print("\nPlease tell me your name:") f_name = input("First name: ")
l_name = input("Last name: ")
formatted_name = get_formatted_name(f_name, l_name) print("\nHello, " + formatted_name + "!") Nesse exemplo, usamos uma versão simples de get_formatted_name() que não envolve nomes do meio. O laço while pede que o usuário forneça seu nome; além disso, solicitamos o primeiro nome e o sobrenome separadamente ❶.

Porém há um problema com esse laço `while`: não definimos uma condição de saída. Onde devemos colocar uma condição de saída quando pedimos uma série de entradas? Queremos que o usuário seja capaz de sair o mais facilmente possível, portanto cada prompt deve oferecer um modo de fazer isso. A instrução `break` permite um modo simples de sair do laço em qualquer prompt: `def get_formatted_name(first_name, last_name):` """Devolve um

nome completo formatado de modo elegante."""

```
    full_name = first_name + ' ' + last_name return full_name.title()
while True: print("\nPlease tell me your name:") print("(enter 'q' at any
time to quit)")
    f_name = input("First name: ") if f_name == 'q': break
    l_name = input("Last name: ") if l_name == 'q': break
    formatted_name = get_formatted_name(f_name, l_name) print("\nHello, " +
formatted_name + "!") Adicionamos uma mensagem que informa como o usuário
pode sair e então encerramos o laço se o usuário fornecer o valor de
saída em qualquer um dos prompts. Agora o programa continuará saudando as
pessoas até que alguém forneça 'q' em algum dos nomes: Please tell me
your name: (enter 'q' at any time to quit) First name: eric Last name:
matthes
Hello, Eric Matthes!


Please tell me your name: (enter 'q' at any time to quit) First name: q
```

FAÇA VOCÊ MESMO

8.6 – Nomes de cidade: Escreva uma função chamada `city_country()` que aceite o nome de uma cidade e seu país. A função deve devolver uma string formatada assim: "Santiago, Chile"

Chame sua função com pelo menos três pares cidade-país e apresente o valor devolvido.

8.7 – Álbum: Escreva uma função chamada `make_album()` que construa um dicionário descrevendo um álbum musical. A função deve aceitar o nome de um artista e o título de um álbum e deve devolver um dicionário contendo essas duas informações. Use a função para criar três dicionários que representem álbuns diferentes. Apresente cada valor devolvido para mostrar que os dicionários estão armazenando as informações do álbum corretamente.

Acrescente um parâmetro opcional em `make_album()` que permita armazenar o número de faixas em um álbum. Se a linha que fizer a chamada incluir um valor para o número de faixas, acrescente esse valor ao dicionário do álbum. Faça pelo menos uma nova chamada da função incluindo o número de faixas em um álbum.

8.8 – Álbuns dos usuários: Comece com o seu programa do Exercício 8.7. Escreva um laço `while` que permita aos usuários fornecer o nome de um artista e o título de um álbum. Depois que tiver essas informações, chame `make_album()` com as entradas do usuário e apresente o dicionário criado. Lembre-se de incluir um valor de saída no laço `while`.

Passando uma lista para uma função

Com frequência, você achará útil passar uma lista para uma função, seja uma lista de nomes, de números ou de objetos mais complexos, como

dicionários. Se passarmos uma lista a uma função, ela terá acesso direto ao conteúdo dessa lista. Vamos usar funções para que o trabalho com listas seja mais eficiente.

Suponha que tenhamos uma lista de usuários e queremos exibir uma saudação a cada um. O exemplo a seguir envia uma lista de nomes a uma função chamada `greet_users()`, que saúda cada pessoa da lista individualmente: greet_users.py def greet_users(names): """Exibe uma saudação simples a cada usuário da lista."""

```
    for name in names: msg = "Hello, " + name.title() + "!"
    print(msg)
❶ usernames = ['hannah', 'ty', 'margot']
  greet_users(usernames) Definimos greet_users() para que espere uma lista
  de nomes, que é armazenada no parâmetro names. A função percorre a lista
  recebida com um laço e exibe uma saudação para cada usuário. Em ❶
  definimos uma lista de usuários e então passamos a lista usernames para
  greet_users() em nossa chamada de função: Hello, Hannah!

  Hello, Ty!
  Hello, Margot!
```

Essa é a saída que queríamos. Todo usuário vê uma saudação personalizada, e você pode chamar a função sempre que quiser para saudar um conjunto específico de usuários.

Modificando uma lista em uma função

Quando passamos uma lista a uma função, ela pode ser modificada. Qualquer alteração feita na lista no corpo da função é permanente, permitindo trabalhar de modo eficiente, mesmo quando lidamos com grandes quantidades de dados.

Considere uma empresa que cria modelos de designs submetidos pelos usuários e que são impressos em 3D. Os designs são armazenados em uma lista e, depois de impressos, são transferidos para uma lista separada. O código a seguir faz isso sem usar funções: printing_models.py # Começa com alguns designs que devem ser impressos unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']

```
  completed_models = []
```

```
# Simula a impressão de cada design, até que não haja mais nenhum #
Transfere cada design para completed_models após a impressão while
unprinted_designs: current_design = unprinted_designs.pop()
  # Simula a criação de uma impressão 3D a partir do design
print("Printing model: " + current_design)
completed_models.append(current_design)
# Exibe todos os modelos finalizados print("\nThe following models have
been printed:") for completed_model in completed_models:
print(completed_model) Esse programa começa com uma lista de designs que
devem ser impressos e uma lista vazia chamada completed_models para a
qual cada design será transferido após a impressão. Enquanto houver
designs em unprinted_designs, o laço while simulará a impressão de cada
um deles removendo um design do final da lista, armazenando-o em
current_design e exibindo uma mensagem informando que o design atual está
sendo impresso. O design então é adicionado à lista de modelos
finalizados. Quando o laço acaba de executar, uma lista de designs
impressos é exibida: Printing model: dodecahedron Printing model: robot
pendant Printing model: iphone case
The following models have been printed: dodecahedron
robot pendant
iphone case
```

Podemos reorganizar esse código escrevendo duas funções, em que cada uma executa uma tarefa específica. A maior parte do código não sofrerá alterações; estamos simplesmente deixando-o mais eficiente. A primeira função tratará a impressão dos designs e a segunda gerará um resumo da impressão feita: ❶ def print_models(unprinted_designs, completed_models): """

```
    Simula a impressão de cada design, até que não haja mais nenhum.
    Transfere cada design para completed_models após a impressão.
    """
    while unprinted_designs: current_design = unprinted_designs.pop()
    # Simula a criação de uma impressão 3D a partir do design
  print("Printing model: " + current_design)
  completed_models.append(current_design)
❷ def show_completed_models(completed_models): """Mostra todos os modelos
impressos."""
    print("\nThe following models have been printed:") for completed_model
  in completed_models: print(completed_model)
  unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
  completed_models = []
```

```
print_models(unprinted_designs, completed_models)
show_completed_models(completed_models) Em ❶ definimos a função
print_models() com dois parâmetros: uma lista de designs a serem
impressos e uma lista de modelos concluídos. Dadas essas duas listas, a
função simula a impressão de cada design esvaziando a lista de designs
não impressos e preenchendo a lista de designs completos. Em ❷ definimos
a função show_completed_models() com um parâmetro: a lista de modelos
finalizados. Dada essa lista, show_completed_models() exibe o nome de
cada modelo impresso.
```

Esse programa produz a mesma saída da versão sem funções, mas o código está muito mais organizado. O código que faz a maior parte do trabalho foi transferido para duas funções separadas, que deixam a parte principal do programa mais fácil de entender. Observe o corpo do programa para ver como é mais simples entender o que esse programa faz: unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']

```
completed_models = []


print_models(unprinted_designs, completed_models)
show_completed_models(completed_models) Definimos uma lista de designs
não impressos e uma lista vazia que armazenará os modelos finalizados.
Então, como já definimos nossas duas funções, tudo que temos a fazer é
chamá-las e passar os argumentos corretos a elas. Chamamos print_models()
e passamos as duas listas de que essa função precisa; conforme esperado,
print_models() simula a impressão dos designs. Em seguida, chamamos
show_completed_models() e passamos uma lista dos modelos concluídos para
que ela possa apresentar os modelos impressos. Os nomes descritivos das
funções permitem que outras pessoas leiam esse código e o entendam, mesmo
que não haja comentários.
```

Esse programa é mais fácil de ser estendido e mantido que a versão sem funções. Se precisarmos imprimir mais designs depois, poderemos simplesmente chamar `print_models()` novamente. Se percebermos que o código de impressão precisa ser modificado, podemos alterar o código uma vez, e nossas alterações estarão presentes em todos os lugares em que a função for chamada. Essa técnica é mais eficiente que ter de atualizar um código separadamente em vários pontos do programa.

Esse exemplo também mostra a ideia de que toda função deve ter uma tarefa específica. A primeira função imprime cada design, enquanto a segunda mostra os modelos concluídos. Isso é mais vantajoso que usar uma única função para executar as duas tarefas. Se você estiver

escrevendo uma função e perceber que ela está fazendo muitas tarefas diferentes, experimente dividir o código em duas funções. Lembre-se de que você sempre pode chamar uma função a partir de outras funções, o que pode ser conveniente quando dividimos uma tarefa complexa em uma série de passos.

Evitando que uma função modifique uma lista

Às vezes, você vai querer evitar que uma função modifique uma lista. Por exemplo, suponha que você comece com uma lista de designs não impressos e escreva uma função que transfira esses designs para uma lista de modelos terminados, como no exemplo anterior. Talvez você decida que, apesar de ter imprimido todos os designs, vai querer manter a lista original de designs não impressos em seus registros. Porém, como você transferiu todos os nomes de designs de `unprinted_designs`, a lista agora está vazia, e essa é a única versão da lista que você tem; a lista original se perdeu. Nesse caso, podemos tratar esse problema passando uma cópia da lista para a função, e não a lista original. Qualquer alteração que a função fizer na lista afetará apenas a cópia, deixando a lista original intacta.

Você pode enviar uma cópia de uma lista para uma função assim: *`nome_da_função`*(*`nome_da_lista`*[:]) A notação de fatia `[:]` cria uma cópia da lista para ser enviada à função. Se não quiséssemos esvaziar a lista de designs não impressos em *print_models.py*, chamaríamos `print_models()` desta maneira: print_models(unprinted_designs[:], completed_models) A função `print_models()` pode fazer seu trabalho, pois ela continua recebendo os nomes de todos os designs não impressos. Porém, dessa vez, ela usa uma cópia da lista original de designs não impressos, e não a lista `unprinted_designs` propriamente dita. A lista `completed_models` será preenchida com os nomes dos modelos impressos, como antes, mas a lista original de designs não impressos não será afetada pela função.

Apesar de poder preservar o conteúdo de uma lista passando uma cópia dela para suas funções, você deve passar a lista original para as funções, a menos que tenha um motivo específico para passar uma cópia. Para uma função, é mais eficiente trabalhar com uma lista existente a fim de evitar o uso de tempo e de memória necessários para

criar uma cópia separada, em especial quando trabalhamos com listas grandes.

FAÇA VOCÊ MESMO

8.9 – Mágicos: Crie uma lista de nomes de mágicos. Passe a lista para uma função chamada `show_magicians()` que exiba o nome de cada mágico da lista.

8.10 – Grandes mágicos: Comece com uma cópia de seu programa do Exercício 8.9. Escreva uma função chamada `make_great()` que modifique a lista de mágicos acrescentando a expressão *o Grande* ao nome de cada mágico. Chame `show_magicians()` para ver se a lista foi realmente modificada.

8.11 – Mágicos inalterados: Comece com o trabalho feito no Exercício 8.10. Chame a função `make_great()` com uma cópia da lista de nomes de mágicos. Como a lista original não será alterada, devolva a nova lista e armazene-a em uma lista separada. Chame `show_magicians()` com cada lista para mostrar que você tem uma lista de nomes originais e uma lista com a expressão *o Grande* adicionada ao nome de cada mágico.

Passando um número arbitrário de argumentos

Às vezes, você não saberá com antecedência quantos argumentos uma função deve aceitar. Felizmente, Python permite que uma função receba um número arbitrário de argumentos da instrução de chamada.

Por exemplo, considere uma função que prepare uma pizza. Ela deve aceitar vários ingredientes, mas não é possível saber com antecedência quantos ingredientes uma pessoa vai querer. A função no próximo exemplo tem um parâmetro `*toppings`, mas esse parâmetro agrupa tantos argumentos quantos forem fornecidos na linha de chamada: pizza.py def make_pizza(*toppings): """Exibe a lista de ingredientes pedidos."""

```
    print(toppings)
make_pizza('pepperoni') make_pizza('mushrooms', 'green peppers', 'extra
cheese') O asterisco no nome do parâmetro *toppings diz a Python para
criar uma tupla vazia chamada toppings e reunir os valores recebidos
nessa tupla. A instrução print no corpo da função gera uma saída que
mostra que Python é capaz de tratar uma chamada de função com um valor e
outra chamada com três valores. As chamadas são tratadas de modo
semelhante. Observe que Python agrupa os argumentos em uma tupla, mesmo
que a função receba apenas um valor: ('pepperoni',)
```

```
('mushrooms', 'green peppers', 'extra cheese') Podemos agora substituir a
instrução print por um laço que percorra a lista de ingredientes e
descreva a pizza sendo pedida: def make_pizza(*toppings): """Apresenta a
pizza que estamos prestes a preparar."""

  print("\nMaking a pizza with the following toppings:") for topping in
toppings: print("- " + topping)
make_pizza('pepperoni') make_pizza('mushrooms', 'green peppers', 'extra
cheese') A função responde de forma apropriada, independentemente de
receber um ou três valores: Making a pizza with the following toppings: -
pepperoni


Making a pizza with the following toppings: - mushrooms
- green peppers
- extra cheese
```

Essa sintaxe funciona, não importa quantos argumentos a função receba.

Misturando argumentos posicionais e arbitrários

Se quiser que uma função aceite vários tipos de argumentos, o parâmetro que aceita um número arbitrário de argumentos deve ser colocado por último na definição da função. Python faz a correspondência de argumentos posicionais e nomeados antes, e depois agrupa qualquer argumento remanescente no último parâmetro.

Por exemplo, se a função tiver que aceitar um tamanho para a pizza, esse parâmetro deve estar antes do parâmetro `*toppings`: def make_pizza(size, *toppings): """Apresenta a pizza que estamos prestes a preparar."""

```
  print("\nMaking a " + str(size) +
  "-inch pizza with the following toppings:") for topping in toppings:
print("- " + topping)
make_pizza(16, 'pepperoni') make_pizza(12, 'mushrooms', 'green peppers',
'extra cheese') Na definição da função, Python armazena o primeiro valor
recebido no parâmetro size. Todos os demais valores que vierem depois são
armazenados na tupla toppings. As chamadas da função incluem um argumento
para o tamanho antes, seguido de tantos ingredientes quantos forem
necessários.
```

Agora cada pizza tem um tamanho e alguns ingredientes, e cada informação é exibida no lugar apropriado, mostrando o tamanho antes e os ingredientes depois: Making a 16-inch pizza with the following toppings: - pepperoni

```
Making a 12-inch pizza with the following toppings: - mushrooms
- green peppers
- extra cheese
```

Usando argumentos nomeados arbitrários

Às vezes, você vai querer aceitar um número arbitrário de argumentos, mas não saberá com antecedência qual tipo de informação será passado para a função. Nesse caso, podemos escrever funções que aceitem tantos pares chave-valor quantos forem fornecidos pela instrução que faz a chamada. Um exemplo envolve criar perfis de usuários: você sabe que obterá informações sobre um usuário, mas não tem certeza quanto ao tipo de informação que receberá. A função `build_profile()` no próximo exemplo sempre aceita um primeiro nome e um sobrenome, mas aceita também um número arbitrário de argumentos nomeados: user_profile.py def build_profile(first, last, **user_info): """Constrói um dicionário contendo tudo que sabemos sobre um usuário."""

```
    profile = {}
❶ profile['first_name'] = first profile['last_name'] = last ❷ for key,
value in user_info.items(): profile[key] = value return profile
  user_profile = build_profile('albert', 'einstein', location='princeton',
  field='physics') print(user_profile) A definição de build_profile()
  espera um primeiro nome e um sobrenome e permite que o usuário passe
  tantos pares nome-valor quantos ele quiser. Os asteriscos duplos antes do
  parâmetro **user_info fazem Python criar um dicionário vazio chamado
  user_info e colocar quaisquer pares nome-valor recebidos nesse
  dicionário. Nessa função, podemos acessar os pares nome-valor em
  user_info como faríamos com qualquer dicionário.
```

No corpo de `build_profile()`, criamos um dicionário vazio chamado `profile` para armazenar o perfil do usuário. Em ❶ adicionamos o primeiro nome e o sobrenome nesse dicionário porque sempre receberemos essas duas informações do usuário. Em ❷ percorremos os pares chave-valor adicionais do dicionário `user_info` e adicionamos cada par ao dicionário `profile`. Por fim, devolvemos o dicionário `profile` à linha que chamou a função.

Chamamos `build_profile()` passando o primeiro nome `'albert'`, o sobrenome `'einstein'` e os dois pares chave-valor `location='princeton'` e `field='physics'`. Armazenamos o dicionário `profile` devolvido em `user_profile` e exibimos o valor dessa variável: {'first_name': 'albert',

'last_name': 'einstein', 'location': 'princeton', 'field': 'physics'}

O dicionário devolvido contém o primeiro nome e o sobrenome do usuário e, nesse caso, a localidade e o campo de estudo também. A função será apropriada, não importa quantos pares chave-valor adicionais sejam fornecidos na chamada da função.

Podemos misturar valores posicionais, nomeados e arbitrários de várias maneiras diferentes quando escrevermos nossas próprias funções. É conveniente saber que todos esses tipos de argumento existem, pois você os verá com frequência quando começar a ler códigos de outras pessoas. Aprender a usar os diferentes tipos corretamente e saber quando usar cada um exige prática. Por enquanto, lembre-se de usar a abordagem mais simples possível, que faça o trabalho necessário. À medida que progredir, você aprenderá a usar a abordagem mais eficiente a cada vez.

FAÇA VOCÊ MESMO

8.12 – Sanduíches: Escreva uma função que aceite uma lista de itens que uma pessoa quer em um sanduíche. A função deve ter um parâmetro que agrupe tantos itens quantos forem fornecidos pela chamada da função e deve apresentar um resumo do sanduíche pedido. Chame a função três vezes usando um número diferente de argumentos a cada vez.

8.13 – Perfil do usuário: Comece com uma cópia de *user_profile.py,* da página 210. Crie um perfil seu chamando `build_profile()`, usando seu primeiro nome e o sobrenome, além de três outros pares chave-valor que o descrevam.

8.14 – Carros: Escreva uma função que armazene informações sobre um carro em um dicionário. A função sempre deve receber o nome de um fabricante e um modelo. Um número arbitrário de argumentos nomeados então deverá ser aceito. Chame a função com as informações necessárias e dois outros pares nome-valor, por exemplo, uma cor ou um opcional. Sua função deve ser apropriada para uma chamada como esta: car = make_car('subaru', 'outback', color='blue', tow_package=True) Mostre o dicionário devolvido para garantir que todas as informações foram armazenadas corretamente.

Armazenando suas funções em módulos

Uma vantagem das funções é a maneira como elas separam blocos de código de seu programa principal. Ao usar nomes descritivos para suas funções, será bem mais fácil entender o seu programa principal. Você pode dar um passo além armazenando suas funções em um arquivo separado chamado *módulo* e, então, *importar* esse módulo em seu

programa principal. Uma instrução `import` diz a Python para deixar o código de um módulo disponível no arquivo de programa em execução no momento.

Armazenar suas funções em um arquivo separado permite ocultar os detalhes do código de seu programa e se concentrar na lógica de nível mais alto. Também permite reutilizar funções em muitos programas diferentes. Quando armazenamos funções em arquivos separados, podemos compartilhar esses arquivos com outros programadores sem a necessidade de compartilhar o programa todo. Saber como importar funções também possibilita usar bibliotecas de funções que outros programadores escreveram.

Há várias maneiras de importar um módulo e vou mostrar cada uma delas rapidamente.

Importando um módulo completo

Para começar a importar funções, inicialmente precisamos criar um módulo. Um *módulo* é um arquivo terminado em *.py* que contém o código que queremos importar para o nosso programa. Vamos criar um módulo que contenha a função `make_pizza()`. Para criar esse módulo, removeremos tudo que está no arquivo *pizza.py*, exceto a função `make_pizza()`: pizza.py def make_pizza(size, *toppings): """Apresenta a pizza que estamos prestes a preparar."""

```
    print("\nMaking a " + str(size) +
    "-inch pizza with the following toppings:") for topping in toppings:
  print("- " + topping) Agora criaremos um arquivo separado chamado
  making_pizzas.py no mesmo diretório em que está pizza.py. Esse arquivo
  importa o módulo que acabamos de criar e, em seguida, faz duas chamadas
  para make_pizza(): making_pizzas.py import pizza


❶ pizza.make_pizza(16, 'pepperoni') pizza.make_pizza(12, 'mushrooms',
'green peppers', 'extra cheese') Quando Python lê esse arquivo, a linha
import pizza lhe diz para abrir o arquivo pizza.py e copiar todas as
funções dele para esse programa. Você não vê realmente o código sendo
copiado entre os arquivos porque Python faz isso internamente, à medida que
o programa executa. Tudo que você precisa saber é que qualquer função
definida em pizza.py agora estará disponível em making_pizzas.py.
```

Para chamar uma função que está em um módulo importado, forneça o nome do módulo, que é `pizza` nesse caso, seguido do nome da função,

`make_pizza()`, separados por um ponto ❶. Esse código gera a mesma saída que o programa original, que não importava um módulo: Making a 16-inch pizza with the following toppings: - pepperoni

```
Making a 12-inch pizza with the following toppings: - mushrooms
- green peppers
- extra cheese
```

Essa primeira abordagem à importação, em que simplesmente escrevemos `import` seguido do nome do módulo, deixa todas as funções do módulo disponíveis ao seu programa. Se você usar esse tipo de instrução `import` para importar um módulo completo chamado *nome_do_módulo.py*, todas as funções do módulo estarão disponíveis por meio da sintaxe a seguir: *`nome_do_módulo.nome_da_função`*()

Importando funções específicas

Podemos também importar uma função específica de um módulo. Eis a sintaxe geral para essa abordagem: from *`nome_do_módulo`* import *`nome_da_função`* Você pode importar quantas funções quiser de um módulo separando o nome de cada função com uma vírgula: from *`nome_do_módulo`* import *`função_0`*, *`função_1`*, *`função_2`*

O exemplo com *making_pizzas.py* teria o seguinte aspecto, se quiséssemos importar somente a função que será utilizada: from pizza import make_pizza

```
make_pizza(16, 'pepperoni') make_pizza(12, 'mushrooms', 'green peppers',
'extra cheese') Com essa sintaxe não precisamos usar a notação de ponto
ao chamar uma função. Como importamos explicitamente a função
make_pizza() na instrução import, podemos chamá-la pelo nome quando ela
for utilizada.
```

Usando a palavra reservada as para atribuir um alias a uma função

Se o nome de uma função que você importar puder entrar em conflito com um nome existente em seu programa ou se o nome da função for longo, podemos usar um *alias* único e conciso, que é um nome alternativo, semelhante a um apelido para a função. Dê esse apelido especial à função quando importá-la.

A seguir, atribuímos um alias `mp()` para a função `make_pizza()` importando `make_pizza as mp`. A palavra reservada `as` renomeia uma

função usando o alias que você fornecer: from pizza import make_pizza as mp

```
mp(16, 'pepperoni') mp(12, 'mushrooms', 'green peppers', 'extra cheese')
A instrução import no exemplo renomeia a função make_pizza() para mp()
nesse programa. Sempre que quiser chamar make_pizza(), você pode
simplesmente escrever mp() em seu lugar, e Python executará o código de
make_pizza(), ao mesmo tempo que evitará confusão com outra função
make_pizza() que você possa ter escrito nesse arquivo de programa.
```

A sintaxe geral para fornecer um alias é: from *`nome_do_módulo`* import *`nome_da_função`* as *`nf`*

Usando a palavra reservada as para atribuir um alias a um módulo

Também podemos fornecer um alias para um nome de módulo. Dar um alias conciso a um módulo, por exemplo, `p` para `pizza`, permite chamar mais rapidamente as funções do módulo. Chamar `p.make_pizza()` é mais compacto que chamar `pizza.make_pizza()`: import pizza as p

```
p.make_pizza(16, 'pepperoni') p.make_pizza(12, 'mushrooms', 'green
peppers', 'extra cheese') O módulo pizza recebe o alias p na instrução
import, mas todas as funções do módulo preservam seus nomes originais.
Chamar as funções escrevendo p.make_pizza() não só é mais compacto que
escrever pizza.make_pizza() como também desvia sua atenção do nome do
módulo e permite dar enfoque aos nomes descritivos de suas funções. Esses
nomes de função, que claramente informam o que cada função faz, são mais
importantes para a legibilidade de seu código que usar o nome completo do
módulo.
```

A sintaxe geral para essa abordagem é: import *`nome_do_módulo`* as *`nm`*

Importando todas as funções de um módulo

Podemos dizer a Python para importar todas as funções de um módulo usando o operador asterisco (`*`): from pizza import *

```
make_pizza(16, 'pepperoni') make_pizza(12, 'mushrooms', 'green peppers',
'extra cheese') O asterisco na instrução import diz a Python para copiar
todas as funções do módulo pizza para esse arquivo de programa. Como
todas as funções são importadas, podemos chamar qualquer função pelo
nome, sem usar a notação de ponto. No entanto, é melhor não usar essa
abordagem quando trabalhar com módulos maiores, que não foram escritos
por você: se o módulo tiver um nome de função que seja igual a um nome
existente em seu projeto, você poderá ter alguns resultados inesperados.
Python poderá ver várias funções ou variáveis com o mesmo nome e, em vez
de importar todas as funções separadamente, ele as sobrescreverá.
```

A melhor abordagem é importar a função ou as funções que você quiser ou importar o módulo todo e usar a notação de ponto. Isso resulta em um código claro, mais fácil de ler e de entender. Incluí esta seção para que você possa reconhecer instruções `import` como esta quando as vir no código de outras pessoas: from *`nome_do_módulo`* import *

Estilizando funções

Você precisa ter alguns detalhes em mente quando estilizar funções. As funções devem ter nomes descritivos, e esses nomes devem usar letras minúsculas e underscores. Nomes descritivos ajudam você e outras pessoas a entenderem o que seu código está tentando fazer. Os nomes de módulos devem usar essas convenções também.

Toda função deve ter um comentário que explique o que ela faz de modo conciso. Esse comentário deve estar imediatamente após a definição da função e deve utilizar o formato de docstring. Se uma função estiver bem documentada, outros programadores poderão usá-la apenas lendo a descrição da docstring. Eles poderão crer que o código funcionará conforme descrito e, desde que saibam o nome da função, os argumentos necessários e o tipo de valor que ela devolve, deverão ser capazes de usá-la em seus programas.

Se você especificar um valor default para um parâmetro, não deve haver espaços em nenhum dos lados do sinal de igualdade: def *`nome_da_função`*(*`parâmetro_0`*, *`parâmetro_1`*='*`valor default`*') A mesma convenção deve ser usada para argumentos nomeados em chamadas de função: *`nome_da_função`*(*`valor_0`*, *`parâmetro_1`*='*`valor`*') A PEP 8 (*https://www.python.org/dev/peps/pep-0008/*) recomenda limitar as linhas de código em 79 caracteres para que todas as linhas permaneçam

visíveis em uma janela de editor com um tamanho razoável. Se um conjunto de parâmetros fizer com que a definição de uma função ultrapasse 79 caracteres, tecle ENTER após o parêntese de abertura na linha da definição. Na próxima linha, tecle TAB duas vezes para separar a lista de argumentos do corpo da função, que estará indentado com apenas um nível.

A maioria dos editores fará automaticamente o alinhamento de qualquer parâmetro adicional para que corresponda à indentação que você determinar na primeira linha: def *nome_da_função*(

```
    parâmetro_0, parâmetro_1, parâmetro_2, parâmetro_3, parâmetro_4,
parâmetro_5): corpo da função...
```

Se seu programa ou módulo tiver mais de uma função, você poderá separá-las usando duas linhas em branco para facilitar ver em que lugar uma função termina e a próxima começa.

Todas as instruções `import` devem estar no início de um arquivo. A única exceção ocorre quando você usa comentários no início de seu arquivo para descrever o programa como um todo.

FAÇA VOCÊ MESMO

8.15 – Impressão de modelos: Coloque as funções do exemplo *print_models.py* em um arquivo separado de nome *printing_functions.py*. Escreva uma instrução `import` no início de *print_models.py* e modifique o arquivo para usar as funções importadas.

8.16 – Importações: Usando um programa que você tenha escrito e que contenha uma única função, armazene essa função em um arquivo separado. Importe a função para o arquivo principal de seu programa e chame-a usando cada uma das seguintes abordagens: import *nome_do_módulo* from *nome_do_módulo* import *nome_da_função* from *nome_do_módulo* import *nome_da_função* as *nf* import *nome_do_módulo* as *nm* from *nome_do_módulo* import *

8.17 – Estilizando funções: Escolha quaisquer três programas que você escreveu neste capítulo e garanta que estejam de acordo com as diretrizes de estilo descritas nesta seção.

Resumo

Neste capítulo aprendemos a escrever funções e a passar argumentos de modo que suas funções tenham acesso às informações necessárias para realizarem sua tarefa. Vimos como usar argumentos posicionais e

nomeados, e aprendemos a aceitar um número arbitrário de argumentos. Conhecemos funções que exibem uma saída e funções que devolvem valores. Aprendemos a usar funções com listas, dicionários, instruções `if` e laços `while`. Também vimos como armazenar suas funções em arquivos separados chamados de *módulos*; desse modo, seus arquivos de programa serão mais simples e mais fáceis de entender. Por fim, aprendemos a estilizar as funções para que seus programas continuem bem estruturados e o mais fácil possível para você e outras pessoas lerem.

Um de nossos objetivos como programador deve ser escrever um código simples, que faça o que você quer, e as funções ajudam nessa tarefa. Elas permitem escrever blocos de código e deixá-los de lado depois que você souber que eles funcionam. Quando souber que uma função executa sua tarefa de forma correta, você poderá se sentir seguro de que ela continuará a funcionar, e poderá prosseguir para a próxima tarefa de programação.

As funções permitem escrever um código uma vez e então reutilizá-lo quantas vezes você quiser. Quando tiver que executar o código de uma função, tudo que você precisa fazer é escrever uma chamada de uma linha, e a função fará o seu trabalho. Quando for necessário modificar o comportamento de uma função, basta modificar apenas um bloco de código e sua alteração terá efeito em todos os lugares em que uma chamada dessa função for feita.

Usar funções deixa seus programas mais fáceis de ler, e bons nomes de função sintetizam o que cada parte de um programa faz. Ler uma série de chamadas de função possibilita ter rapidamente uma noção do que um programa faz, se comparado à leitura de uma série longa de blocos de código.

As funções também deixam seu código mais fácil de testar e de depurar. Quando a maior parte do trabalho de seu programa for feita por um conjunto de funções, em que cada uma tem uma tarefa específica, será muito mais fácil testar e dar manutenção no código que você escreveu. É possível escrever um programa separado que chame cada função e testar se ela funciona em todas as situações com as quais você poderá se deparar. Ao fazer isso, você se sentirá mais seguro de que suas funções estarão corretas sempre que forem chamadas.

No Capítulo 9 aprenderemos a escrever classes. As classes combinam funções e dados em um pacote organizado, que pode ser usado de maneiras flexíveis e eficientes.