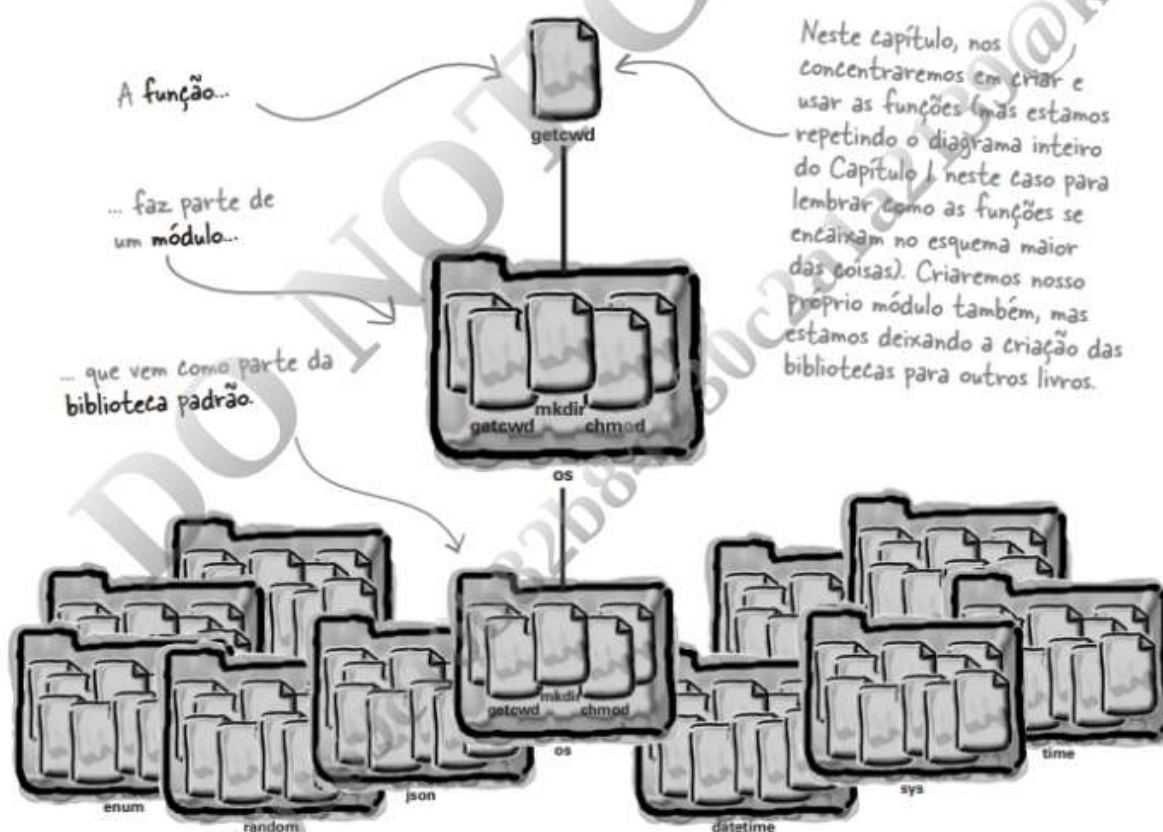


Reutilizando o Código com Funções

Embora algumas linhas de código possam fazer muita coisa no Python, mais cedo ou mais tarde você descobrirá que a base de código do seu programa está aumentando... e, quando aumenta, a dificuldade de gerenciamento aumenta rapidamente. O que começou com 20 linhas de código em Python aumentou para 500 linhas ou mais! Quando isso acontece, é hora de começar a pensar em quais estratégias você poderá usar para reduzir a complexidade de sua base de código:

Como muitas outras linguagens de programação, o Python suporta a **modularidade**, no sentido de que você pode dividir as grandes partes de código em pedaços menores e mais gerenciáveis. Você faz isso criando **funções**, que podem ser consideradas como partes nomeadas do código. Lembre-se do diagrama do Capítulo 1, que mostra a relação entre as funções, os módulos e a biblioteca padrão:



Neste capítulo, nos concentraremos no que está envolvido na criação de suas próprias funções, mostradas bem no topo do diagrama. Assim que você ficar satisfeito com a criação das funções, também mostraremos como criar um módulo.

Apresentando as Funções

Antes de transformar nosso código existente em uma função, vamos passar um tempo vendo a anatomia de *qualquer* função no Python. Assim que esta introdução for concluída, veremos nosso código existente e executaremos as etapas necessárias para transformá-lo em uma função que você poderá reutilizar.

Não se preocupe com os detalhes ainda. Tudo que você precisa fazer aqui é ter uma ideia de como são as funções no Python, como descrito nesta e na próxima página. Veremos os detalhes de tudo que você precisa saber quando este capítulo avançar. A janela do IDLE nesta página apresenta um modelo que você pode usar ao criar qualquer função. Como é possível ver, considere o seguinte:

1 As funções introduzem duas palavras-chave novas: `def` e `return`

Essas palavras-chaves têm uma cor diferente no IDLE. A palavra-chave `def` nomeia a função e detalha os argumentos que a função pode ter. O uso da palavra-chave `return` é opcional, e ela é usada para passar de volta um valor para o código que chamou a função.

2 As funções podem aceitar dados de argumento

Uma função pode aceitar os dados de argumento (ou seja, a entrada para a função). Você pode especificar uma lista de argumentos entre parênteses na linha `def`, após o nome da função.

3 As funções têm código e (geralmente) documentação

O código é recuado um nível abaixo da linha `def` e deve incluir comentários onde fazem sentido. Demonstramos dois modos de adicionar comentários ao código: usando uma string com três aspas (conhecida como **docstring**) e usando um comentário com uma linha, prefixado com o símbolo `#` (e mostrado abaixo).

A linha "`def`" nomeia a função e lista qualquer argumento.

"docstring" descreve a finalidade da função.

```
function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)
def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value
```

Um modelo de função "útil"

Seu código fica aqui (no lugar dos espaços reservados a comentários com uma linha).



C\u00f3digo S\u00e9rio

O Python usa o nome "fun\u00e7\u00e3o" para descrever uma parte reutiliz\u00e1vel de c\u00f3digo. Outras linguagens de programa\u00e7\u00e3o usam nomes como "procedimento", "sub-rotina" e "m\u00e9todo". Quando uma fun\u00e7\u00e3o faz parte de uma classe do Python, \u00e9 conhecida como "m\u00e9todo". Voc\u00ea aprender\u00e1 sobre as classes e m\u00e9todos do Python mais tarde no cap\u00edtulo.

e o tipo?

E as Informações do Tipo?

Veja de novo nosso exemplo de função. Exceto pelo código a executar, você acha que está faltando algo? Existe algo que você esperaria que fosse especificado, mas não foi? Dê outra olhada:

```
function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value

Ln: 8 Col: 0
```

Existe algo que falta no modelo desta função?

Estou com um pouco de medo do modelo dessa função. Como o interpretador sabe quais são os tipos dos argumentos, assim como qual é o tipo do valor de retorno?



Ele não sabe, mas não deixe que isso o preocupe.

O interpretador Python não o força a especificar o tipo dos argumentos de sua função ou o valor de retorno. Dependendo das linguagens de programação usadas antes, isso pode assustar um pouco. Não deixe que isso aconteça.

O Python permite que você envie qualquer *objeto* como um argumento e passe de volta qualquer *objeto* como um valor de retorno. O interpretador não se importa nem verifica o tipo desses objetos (apenas que eles são fornecidos).

Com o Python 3 é possível *indicar* os tipos esperados dos argumentos/valores de retorno, e faremos exatamente isso mais tarde no capítulo. Contudo, indicar os tipos esperados não ativa “magicamente” a verificação do tipo, pois o Python *nunca* verifica os tipos dos argumentos nem dos valores de retorno.

Nomeando uma Parte do Código com "def"

Assim que você tiver identificado a parte do código do Python que deseja reutilizar, será hora de criar uma função. Você cria uma função usando a palavra-chave `def` (que é uma abreviação de *define*). A palavra-chave `def` é seguida do nome da função, uma lista de argumentos vazia e opcional (entre parênteses), dois-pontos e uma ou mais linhas de código recuado.

Lembre-se do programa `vowels7.py` no final do último capítulo, que, dada uma palavra, imprime as vogais contidas nessa palavra:

Pegue um conjunto de vogais...
... e uma palavra
... e então faça uma interseção.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Isto é "`vowels7.py`" no final do Capítulo 3.

← Exiba qualquer resultado.

Imaginemos que você pretende usar estas cinco linhas de código muitas vezes em um programa muito maior. A última coisa que você deseja fazer é copiar e colar o código sempre que necessário... portanto, para manter as coisas gerenciáveis e assegurar que precisará manter apenas **uma cópia** desse código, vamos criar uma função.

Demonstraremos como no Python Shell (por ora). Para transformar as cinco linhas de código acima em uma função, use a palavra-chave `def` para indicar que uma função está iniciando, dê um nome descritivo à função (*sempre* uma boa ideia), forneça uma lista de argumentos vazia e opcional entre parênteses, seguida de dois-pontos, e então recue as linhas de código relativas à palavra-chave `def`, como a seguir:

Reserve um tempo para escolher um bom nome descritivo para sua função.

Inicie com a palavra-chave "`def`".

Dê a sua função um belo nome descritivo.

Forneça uma lista de argumentos opcional — neste caso, a função não tem argumentos, portanto, a lista está vazia.

```
>>> def search4vowels():
    vowels = set('aeiou')
    word = input("Provide a word to search for vowels: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

← Não se esqueça dos dois-pontos.

As cinco linhas de código do programa "`vowels7.py`" adequadamente recuadas

← Como este é o shell, lembre-se de pressionar a tecla Enter **DUAS VEZES** para confirmar que o código recuado terminou.

Agora que a função existe, vamos chamá-la, para ver se está funcionando como esperamos.

chamando as funções

Chamando Sua Função

Para chamar as funções no Python, forneça o nome da função com os valores para qualquer argumento que a função espera. Como a função `search4vowels` (atualmente) não tem argumentos, podemos chamá-la com uma lista de argumentos vazia, assim:

```
>>> search4vowels()
Provide a word to search for vowels: hitch-hiker
e
i
```

Chamar a função novamente a executa de novo:

```
>>> search4vowels()
Provide a word to search for vowels: galaxy
a
```

Nenhuma surpresa aqui: chamar a função executa seu código.

Edite sua função em um editor, não no prompt

No momento, o código da função `search4vowels` foi fornecido no prompt `>>>` e fica assim:

```
>>> def search4vowels():
    vowels = set('aeiou')
    word = input("Provide a word to search for vowels: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Nossa função
como fornecida no
prompt do shell.

Para trabalhar mais com este código, você pode chamá-lo de novo no prompt `>>>` e editá-lo, mas isso logo se torna muito chato. Lembre-se de que assim que o código com o qual você está trabalhando no prompt `>>>` tiver mais de algumas linhas, será melhor copiá-lo para uma janela de edição do IDLE. É possível editá-lo com muito mais facilidade nela. Portanto, vamos fazer isso antes de continuar.

Crie uma nova janela de edição do IDLE vazia, e então copie o código da função do prompt `>>>` (sem copiar os caracteres `>>>`) e cole-o na janela de edição. Assim que ficar satisfeito com a formatação e o recuo estiver correto, salve o arquivo como `vsearch.py` antes de continuar.

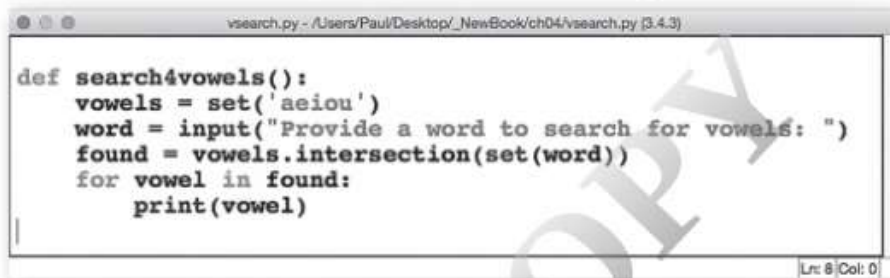
**Salve seu código
como "vsearch.py"
depois de copiar
o código da
função no shell.**

reutilização do código

Use o Editor do IDLE para Fazer Alterações

Veja como fica o arquivo `vsearch.py` no IDLE:

Agora o código da função está em uma janela de edição do IDLE e foi salvo como "`vsearch.py`".



```
def search4vowels():  
    vowels = set('aeiou')  
    word = input("Provide a word to search for vowels: ")  
    found = vowels.intersection(set(word))  
    for vowel in found:  
        print(vowel)
```

Se você pressionar F5 na janela de edição, acontecerão duas coisas: o shell do IDLE virá para o primeiro plano e reiniciará. Contudo, nada aparece na tela. Tente isso agora para ver o que queremos dizer: pressione F5.

O motivo para nada ser exibido é que você ainda tem que chamar a função. Iremos chamá-la daqui a pouco, mas agora faremos uma alteração em nossa função antes de continuar. É uma pequena alteração, mas importante.

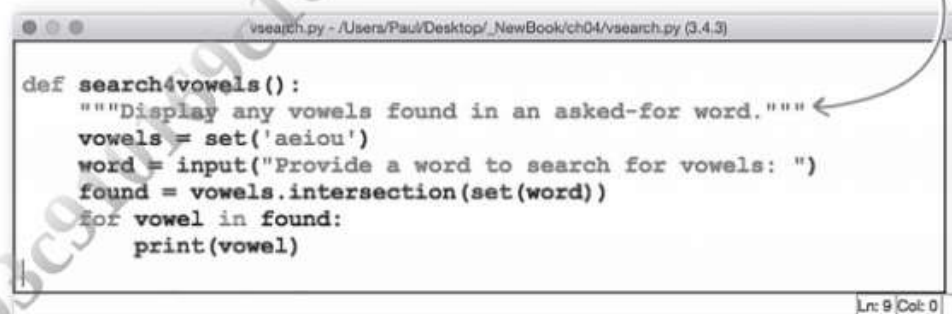
Adicionaremos uma documentação no início de nossa função.

Para adicionar um comentário com várias linhas (**docstring**) a qualquer código, coloque o texto do comentário entre aspas triplas.

Veja o arquivo `vsearch.py` de novo, com uma docstring adicionada ao início da função. Vá em frente e faça a alteração no código também:

Se o IDLE exibir um erro quando você pressionar F5, não entre em pânico! Volte para a janela de edição e verifique se seu código está igual ao nosso, e então tente de novo.

Uma docstring foi adicionada ao código da função, que descreve (rapidamente) a finalidade dela.

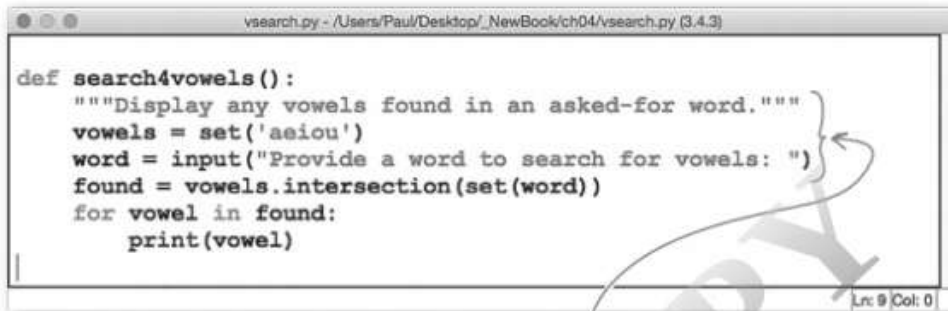


```
def search4vowels():  
    """Display any vowels found in an asked-for word."""  
    vowels = set('aeiou')  
    word = input("Provide a word to search for vowels: ")  
    found = vowels.intersection(set(word))  
    for vowel in found:  
        print(vowel)
```


onde está a conformidade com o PEP

O Que se Passa com Todas Essas Strings?

Veja de novo a função como ela está atualmente. Preste muita atenção nas três strings do código indicadas no IDLE:



```
def search4vowels():  
    """Display any vowels found in an asked-for word."""  
    vowels = set('aeiou')  
    word = input("Provide a word to search for vowels: ")  
    found = vowels.intersection(set(word))  
    for vowel in found:  
        print(vowel)
```

O destaque da sintaxe do IDLE mostra que temos um problema de consistência no uso das aspas da string. Quando usamos qual estilo?

Compreendendo os caracteres de aspas da string

No Python, as strings podem ficar entre aspas simples ('), aspas duplas (") ou entre o que é conhecido como aspas triplas (""" ou ''').

Como mencionado antes, as aspas triplas em torno das strings são conhecidas como **docstrings**, porque são usadas principalmente para documentar a finalidade de uma função (como mostrado acima). Mesmo que você possa usar """ ou ''' em torno das docstrings, a maioria dos programadores Python prefere usar """. As docstrings têm uma característica interessante, no sentido de que podem se estender em várias linhas (outras linguagens de programação usam o nome "heredoc" para o mesmo conceito).

As strings entre aspas simples (') ou duplas (") **não podem** se estender em várias linhas: você deve terminar a string com um caractere de aspa correspondente na mesma linha (pois o Python usa o final da linha como um término de instrução).

Qual caractere você usa em torno de suas strings é com você, embora usar as aspas simples seja muito popular para a maioria dos programadores Python. Mas, principalmente, seu uso deve ser consistente.

O código mostrado no início desta página (apesar de ter apenas algumas linhas de código) *não* é consistente em seu uso das aspas da string. Note que o código é bem executado (pois o interpretador não se importa com o estilo usado), mas misturar e combinar os estilos pode dificultar a leitura do código mais do que o necessário (o que é uma vergonha).

**Seja consistente
no uso das
aspas da string.
Se possível, use
aspas simples.**

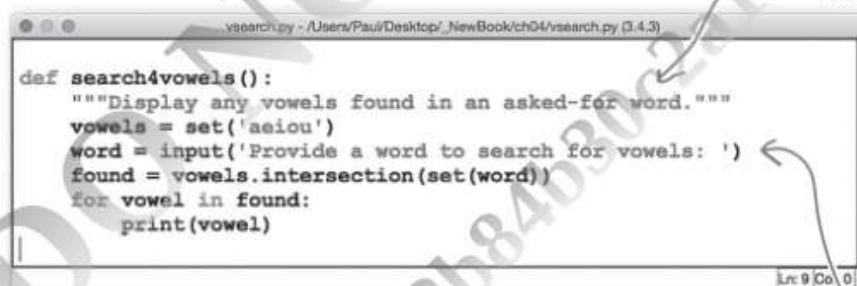
Siga a Melhor Prática Conforme os PEPs

Quanto à formatação do código (não apenas as strings), a comunidade de programação Python passou muito tempo estabelecendo e documentando a melhor prática. Essa melhor prática é conhecida como **PEP 8**. PEP é uma abreviação de "Protocolo de Melhoria do Python" (em inglês: *Python Enhancement Protocol*).

Existem muitos documentos PEP, e eles basicamente detalham as melhorias propostas e implementadas na linguagem de programação Python (sobre o que fazer e não fazer), assim como descrevem vários processos do Python. Os detalhes dos documentos PEP podem ser muito técnicos e (geralmente) complexos. Assim, a grande maioria dos programadores Python sabe de sua existência, mas raramente interage com os PEPs em detalhes. Isso ocorre na maioria dos PEPs, exceto no PEP 8.

O PEP 8 é o guia de estilo para o código do Python. É uma leitura recomendada para todos os programadores Python e é o documento que sugere a "consistência" nas aspas, descrito na última página. Reserve um tempo para ler o PEP 8 pelo menos uma vez. Outro documento, PEP 257, fornece convenções sobre como formatar as docstrings e também vale a pena ser lido.

Veja a função `search4vowels` mais uma vez segundo sua conformidade com o PEP 8 e PEP 257. As alterações não são muitas, mas padronizar as aspas simples em torno de nossas strings (mas não em nossas docstrings) fica muito melhor:



```
def search4vowels():  
    """Display any vowels found in an asked-for word."""  
    vowels = set('aeiou')  
    word = input('Provide a word to search for vowels: ')  
    found = vowels.intersection(set(word))  
    for vowel in found:  
        print(vowel)
```

Encontre a
lista dos PEPs
aqui: [https://
www.python.
org/dev/peps/
\(conteúdo
em inglês\).](https://www.python.org/dev/peps/)

Esta é uma
docstring compatível
com o PEP 257.

Naturalmente, você não precisa escrever um código *exatamente* de acordo com o PEP 8. Por exemplo, o nome da função, `search4vowels`, não segue as diretrizes que sugerem que as palavras no nome de uma função devem ficar separadas por um sublinhado: um nome com mais conformidade seria `search_for_vowels`. Note que o PEP 8 é um conjunto de diretrizes, não uma regra. Você não tem que segui-lo, apenas considerar, e gostamos do nome `search4vowels`.

Dito isso, a grande maioria dos programadores Python agradecerá a você por escrever um código que segue o PEP 8, pois sua leitura é geralmente muito mais fácil do que a do código que não o segue.

Agora vamos melhorar a função `search4vowels` para aceitar os argumentos.

Prestamos
atenção no
conselho do PEP
8 sobre ser
consistente com
as aspas simples
usadas em torno
das strings.

adicione um argumento

As Funções Podem Aceitar Argumentos

Em vez de fazer a função solicitar ao usuário uma palavra para ser pesquisada, vamos mudar a função `search4vowels`, para que possamos passar-lhe a palavra como entrada para um argumento.

Adicionar um argumento é simples: basta inserir o nome do argumento entre parênteses na linha `def`. Então esse nome do argumento se torna uma variável no suíte da função. É uma edição fácil.

Também removeremos a linha de código que solicita ao usuário que forneça uma palavra para ser pesquisada, o que é outra edição fácil.

Vamos lembrar o estado atual de nosso código:

Lembre-se:
"suíte" é o
jargão Python
para "bloco".

Aqui está nossa função original.

```
def search4vowels():  
    """Display any vowels found in an asked-for word."""  
    vowels = set('aeiou')  
    word = input('Provide a word to search for vowels: ')  
    found = vowels.intersection(set(word))  
    for vowel in found:  
        print(vowel)
```

Ln: 9 | Col: 0

Aplicar as duas edições sugeridas (acima) em nossa função resulta na janela de edição do IDLE ficando assim (nota: atualizamos nossa docstring também, o que *sempre* é uma boa ideia):

Esta linha não é mais necessária.

Coloque o nome do argumento entre parênteses.

```
def search4vowels(word):  
    """Display any vowels found in a supplied word."""  
    vowels = set('aeiou')  
    found = vowels.intersection(set(word))  
    for vowel in found:  
        print(vowel)
```

A chamada para a função "input" acabou (pois não precisamos mais dessa linha de código).

Ln: 8 | Col: 0

Salve seu arquivo após cada alteração do código antes de pressionar F5 para levar a nova versão de sua função para dar uma volta.



Test Drive

Com seu código carregado na janela de edição do IDLE (e salvo), pressione F5, e então chame a função algumas vezes e veja o que acontece:

O código
"search4vowels"
atual

```
vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)

Python 3.4.3 Shell

===== RESTART =====
>>>
>>> search4vowels()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    search4vowels()
TypeError: search4vowels() missing 1 required positional argument: 'word'
>>> search4vowels('hitch-hiker')
e
i
>>> search4vowels('hitch-hiker', 'galaxy')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    search4vowels('hitch-hiker', 'galaxy')
TypeError: search4vowels() takes 1 positional argument but 2 were given
>>> |
```

Embora tenhamos chamado a função "search4vowels" três vezes neste Test Drive, a única chamada executada com sucesso foi a que passou um argumento de string. As outras duas falharam. Reserve um momento para ler as mensagens de erro produzidas pelo interpretador para descobrir por que cada chamada incorreta falhou.

^{não existem} Perguntas Idiotas

P: Estou limitado a apenas um argumento ao criar funções no Python?

R: Não, você pode ter quantos argumentos quiser, dependendo do serviço que a função está fornecendo. Estamos iniciando deliberadamente com um exemplo simples, e veremos exemplos mais complicados quando o capítulo avançar. Você pode fazer muito com os argumentos nas funções no Python, e pretendemos analisar grande parte do que é possível nas próximas 12 páginas ou mais.