CAPÍTULO 8

LENDO E ESCREVENDO EM ARQUIVOS

As variáveis são uma maneira conveniente de armazenar dados enquanto seu programa estiver executando, porém, se quiser que seus dados persistam mesmo após o programa ter encerrado, será necessário salvá-los em um arquivo. Podemos pensar no conteúdo de um arquivo como um único valor de string, potencialmente com gigabytes de tamanho. Neste capítulo, aprenderemos a usar o Python para criar, ler e salvar arquivos no disco rígido.

Arquivos e paths de arquivo

Um arquivo tem duas propriedades fundamentais: um *nome de arquivo* e um *path*. O path especifica a localização de um arquivo no computador. Por exemplo, há um arquivo em meu laptop com Windows 7 cujo nome é *projects.docx* no path *C:\Users\asweigart\Documents*. A parte do nome do arquivo após o último ponto é chamada de *extensão* do arquivo e informa o seu tipo. *project.docx* é um documento Word, e *Users*, *asweigart* e *Documents* referem-se a *pastas* (também chamadas de *diretórios*). As pastas podem conter arquivos e outras pastas. Por exemplo, *project.docx* está na pasta *Documents* que está na pasta *asweigart* que está na pasta *Users*. A figura 8.1 mostra a organização dessa pasta.

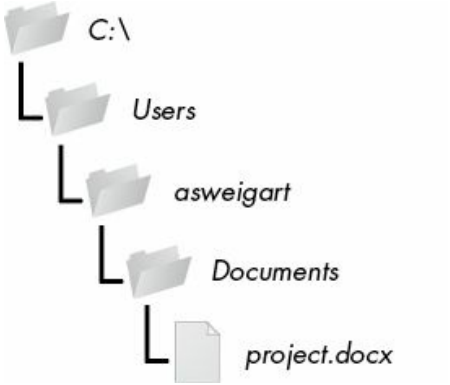

Figura 8.1 – Um arquivo em uma hierarquia de pastas.

A parte referente a *C:* do path é a *pasta-raiz*, que contém todas as demais pastas. No Windows, a pasta-raiz chama-se *C:* e é também chamada de *drive C:*. No OS X e no Linux, a pasta-raiz é */*. Neste livro, usarei a pasta-raiz *C:* no estilo do Windows. Se você estiver inserindo os exemplos do shell interativo no OS X ou no Linux, especifique */* no lugar de *C:*.

Volumes adicionais, como um drive de DVD ou de pen drive USB, aparecerão de forma diferente em sistemas operacionais distintos. No Windows, eles aparecerão como drives-raiz com uma nova letra, por exemplo, *D:* ou *E:*. No OS X, aparecerão como novas pastas dentro da pasta */Volumes*. No Linux, aparecerão como novas pastas dentro da pasta */mnt* ("mount"). Além disso, observe que, enquanto os nomes das pastas e dos arquivos não fazem distinção entre letras maiúsculas e minúsculas no Windows e no OS X, essa diferença existe no Linux.

Barra invertida no Windows e barra para frente no OS X e no Linux

No Windows, os paths são escritos com barras invertidas (\) como separador entre os nomes das pastas. No OS X e no Linux, porém, utilize a barra para frente (/) como separador de path. Se quiser que seus programas funcionem em todos os sistemas operacionais, seus scripts Python deverão ser escritos para tratar ambos os casos.

Felizmente, isso é simples de fazer com a função os.path.join(). Se você lhe passar os valores de string com os nomes individuais dos arquivos e das pastas de seu path, os.path.join() retornará uma string com um path de arquivo que utilizará os separadores corretos de path. Digite o seguinte no shell interativo:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

Estou executando esses exemplos de shell interativo no Windows, portanto os.path.join('usr', 'bin', 'spam') retornou 'usr\\bin\\spam'. (Observe que as barras invertidas estão duplicadas, pois cada barra invertida deve ser escapada por outro caractere de barra invertida.) Se essa função tivesse sido chamada no

OS X ou no Linux, a string seria 'usr/bin/spam'.

A função os.path.join() será útil se houver necessidade de criar strings para os nomes de arquivo. Essas strings serão passadas para diversas funções relacionadas a arquivos que serão apresentadas neste capítulo. O exemplo a seguir une os nomes de uma lista de nomes de arquivo no final do nome de uma pasta:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

Diretório de trabalho atual

Todo programa executado em seu computador tem um *diretório de trabalho atual* (current working directory, ou cwd). Supõe-se que qualquer nome de arquivo ou path que não comece com a pasta-raiz esteja no diretório de trabalho atual. Podemos obter o diretório de trabalho atual como um valor de string usando a função os.getcwd() e alterá-lo com os.chdir(). Digite o seguinte no shell interativo:

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Nesse caso, o diretório de trabalho atual está definido como *C:\Python34*, portanto o nome de arquivo *project.docx* refere-se a *C:\Python34\project.docx*. Quando alteramos o diretório de trabalho atual para *C:\Windows*, *project.docx* é interpretado como *C:\Windows\project.docx*.

O Python exibirá um erro se você tentar mudar para um diretório inexistente.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
```

```
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

NOTA Embora pasta seja o nome mais moderno para diretório, observe que o *diretório de trabalho atual* (ou somente *diretório de trabalho*) é o termo-padrão, e não pasta de trabalho atual.

Comparação entre paths absolutos e relativos

Há duas maneiras de especificar um path de arquivo.

- Um *path absoluto*, que sempre começa com a pasta-raiz.
- Um *path relativo*, que é relativo ao diretório de trabalho atual do programa.

Temos também as pastas *ponto* (.) e *ponto-ponto* (..). Essas não são pastas de verdade, mas nomes especiais que podem ser usados em um path. Um único ponto para um nome de pasta é a forma abreviada de "este diretório". Dois pontos (ponto-ponto) quer dizer "a pasta pai".

A figura 8.2 contém um exemplo de algumas pastas e arquivos. Quando o diretório de trabalho atual estiver definido com *C:\bacon*, os paths relativos das outras pastas e dos arquivos serão definidos como mostra a figura.

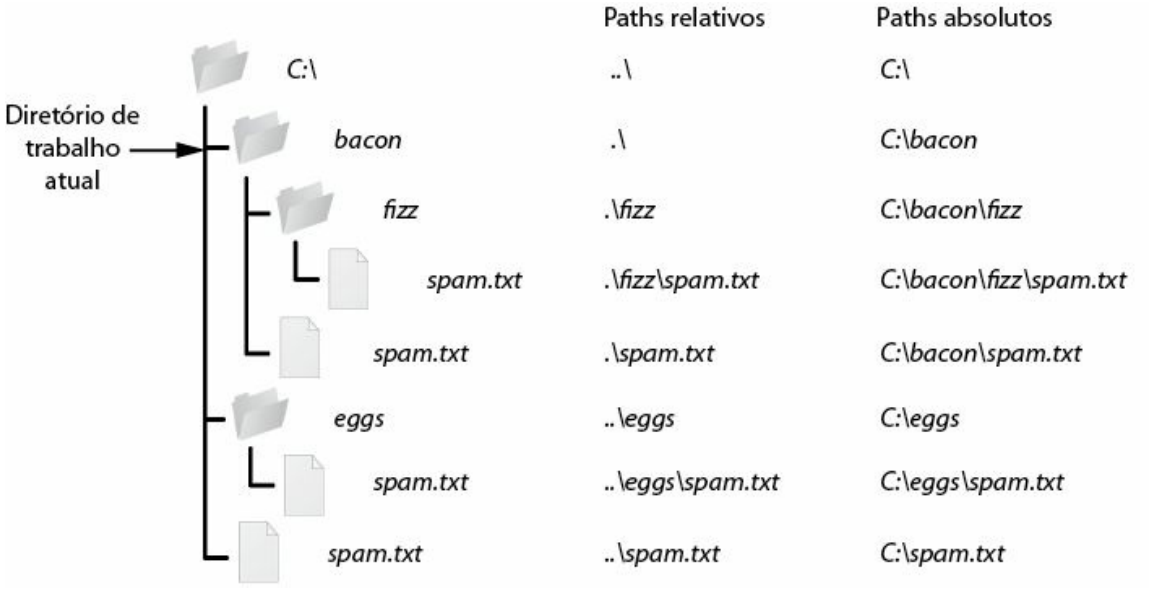

Figura 8.2 – Os paths relativos para as pastas e os arquivos no diretório de trabalho C:\bacon.

O .\ no início de um path relativo é opcional. Por exemplo, *.\spam.txt* e *spam.txt* referem-se ao mesmo arquivo.

Criando novas pastas com os.makedirs()

Seus programas podem criar novas pastas (diretórios) com a função os.makedirs(). Digite o seguinte no shell interativo:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Esse comando criará não só a pasta *C:\delicious* como também uma pasta *walnut* em *C:\delicious* e uma pasta *waffles* em *C:\delicious\walnut*, isto é, os.makedirs() criará qualquer pasta intermediária necessária para garantir que o path completo exista. A figura 8.3 mostra essa hierarquia de pastas.

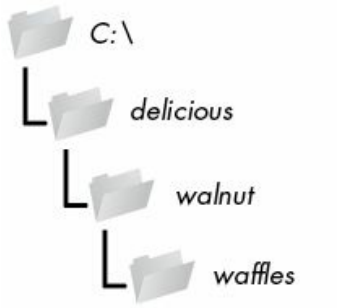

Figura 8.3 – O resultado de os.makedirs('C:\\delicious\\walnut\\waffles').

Módulo os.path

O módulo os.path contém muitas funções úteis relacionadas a nomes e paths de arquivo. Por exemplo, já usamos os.path.join() para criar paths de maneira que isso funcione em qualquer sistema operacional. Como os.path é um módulo do módulo os, podemos importá-lo simplesmente executando import os. Sempre que seus programas precisarem trabalhar com arquivos, pastas ou paths de arquivo, você poderá consultar os pequenos exemplos desta seção. A documentação completa do módulo os.path está no site do Python em *http://docs.python.org/3/library/os.path.html*.

NOTA A maioria dos próximos exemplos nesta seção exigirá o módulo os, portanto lembre-se de importá-lo no início de qualquer script que você criar ou sempre que

reiniciar o IDLE. Caso contrário, uma mensagem de erro NameError: name 'os' is not defined (NameError: nome ‘os’ não está definido) será exibida.

Lidando com paths absolutos e relativos

O módulo os.path disponibiliza funções que retornam o path absoluto de um path relativo e para verificar se um dado path representa um path absoluto.

- Chamar os.path.abspath(*path*) retornará uma string com o path absoluto referente ao argumento. Essa é uma maneira fácil de converter um path relativo em um path absoluto.
- Chamar os.path.isabs(*path*) retornará True se o argumento for um path absoluto e False se for um path relativo.
- Chamar os.path.relpath(*path*, *início*) retornará uma string contendo um path relativo ao path *início* para *path*. Se *início* não for especificado, o diretório de trabalho atual será usado como path de início.

Teste essas funções no shell interativo:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Como *C:\Python34* era o diretório de trabalho quando os.path.abspath() foi chamado, a pasta “ponto” representa o path absoluto 'C:\\Python34'.

NOTA Como seu sistema provavelmente tem arquivos e pastas diferentes em relação ao meu sistema, você não poderá seguir exatamente todos os exemplos deste capítulo. Apesar disso, tente acompanhar usando pastas que existam em seu computador.

Digite as seguintes chamadas a os.path.relpath() no shell interativo:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
```

'C:\\Python34'

Chamar os.path.dirname(*path*) retornará uma string contendo tudo que estiver antes da última barra no argumento path. Chamar os.path.basename(*path*) retornará uma string contendo tudo que estiver após a última barra no argumento path. O nome do diretório e o nome base de um path estão representados na figura 8.4.

Figura 8.4 – O nome base está depois da última barra em um path e corresponde ao nome do arquivo. O nome do diretório corresponde a tudo que estiver antes da última barra.

Por exemplo, digite o seguinte no shell interativo:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

Se o nome de diretório e o nome base de um path forem necessários ao mesmo tempo, você poderá simplesmente chamar os.path.split() para obter um valor de tupla contendo essas duas strings, como em:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Observe que a mesma tupla poderia ter sido criada por meio da chamada a os.path.dirname() e a os.path.basename(), com seus valores de retorno inseridos em uma tupla.

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

Porém os.path.split() será um atalho conveniente se você precisar de ambos os valores.

Além disso, observe que os.path.split() *não* recebe um path de arquivo e retorna uma lista de strings com cada pasta. Para isso, utilize o método de

string split() e separe a string em os.sep. Lembre-se do que vimos anteriormente, que a variável os.sep é definida com a barra correta de separação de pastas para o computador que estiver executando o programa.

Por exemplo, digite o seguinte no shell interativo:

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

Em sistemas OS X e Linux, haverá uma string vazia no início da lista retornada:

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

O método de string split() funciona retornando uma lista contendo cada parte do path. Esse método funcionará em qualquer sistema operacional se os.path.sep lhe for passado.

Obtendo os tamanhos dos arquivos e o conteúdo das pastas

Após ter dominado as maneiras de lidar com paths de arquivo, você poderá começar a reunir informações sobre arquivos e pastas específicos. O módulo os.path disponibiliza funções para obter o tamanho de um arquivo em bytes e os arquivos e as pastas que estiverem em uma determinada pasta.

- Chamar os.path.getsize(*path*) retornará o tamanho em bytes do arquivo no argumento *path*.
- Chamar os.listdir(*path*) retornará uma lista de strings com nomes de arquivo para cada arquivo no argumento *path*. (Observe que essa função está no módulo os, e não em os.path.)

Eis o que obtive quando testei essas funções no shell interativo:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--trecho removido--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Como podemos ver, o programa *calc.exe* em meu computador tem 776.192 bytes e tenho muitos arquivos em *C:\Windows\system32*. Se quiser descobrir o tamanho total de todos os arquivos nesse diretório, poderei usar os.path.getsize() e os.listdir() juntos.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
    totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32',
filename))


>>> print(totalSize)
1117846456
```

À medida que percorro todos os nomes de arquivo da pasta *C:\Windows\System32* em um loop, a variável totalSize é incrementada com o tamanho de cada arquivo. Observe que, quando chamo os.path.getsize(), utilizo os.path.join() para unir o nome da pasta ao nome do arquivo atual. O inteiro que os.path.getsize() retorna é somado ao valor de totalSize. Após percorrer todos os arquivos no loop, apresento totalSize para ver o tamanho total da pasta *C:\Windows\System32*.

Verificando a validade de um path

Muitas funções Python falharão gerando um erro se você lhes fornecer um path inexistente. O módulo os.path disponibiliza funções para verificar se um dado path existe e se é um arquivo ou uma pasta.

- Chamar os.path.exists(*path*) retornará True se o arquivo ou a pasta referenciada no argumento existir e retornará False caso contrário.
- Chamar os.path.isfile(*path*) retornará True se o argumento referente ao path existir e for um arquivo e retornará False caso contrário.
- Chamar os.path.isdir(*path*) retornará True se o argumento referente ao path existir e for uma pasta e retornará False caso contrário.

Eis o que obtive quando testei essas funções no shell interativo:

```
>>> os.path.exists('C:\\Windows')
True
```

```
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

Podemos determinar se há um DVD ou um pen drive conectado ao computador no momento fazendo a verificação com a função os.path.exists(). Por exemplo, se eu quiser verificar se há um pen drive com um volume chamado *D:* em meu computador Windows, posso fazer isso com:

```
>>> os.path.exists('D:\\')
False
```

Opa! Parece que me esqueci de conectar o meu pen drive.

Processo de leitura/escrita

Depois que se sentir à vontade para trabalhar com pastas e paths relativos, você poderá especificar a localização dos arquivos a serem lidos e escritos. As funções discutidas nas próximas seções se aplicam a arquivos em formato texto simples. *Arquivos em formato texto simples* (plaintext files) contêm somente caracteres básicos de texto e não incluem informações sobre fonte, tamanho ou cor. Os arquivos-texto com extensão *.txt* ou arquivos de scripts Python com extensão *.py* são exemplos de arquivos em formato texto simples. Eles podem ser abertos com o aplicativo Notepad do Windows ou com o TextEdit do OS X. Seus programas poderão ler facilmente o conteúdo de arquivos em formato texto simples e tratá-los como um valor normal de string.

Os *arquivos binários* correspondem a todos os demais tipos de arquivos, por exemplo, documentos de processadores de texto, PDFs, imagens, planilhas e programas executáveis. Se você abrir um arquivo binário no Notepad ou no TextEdit, seu conteúdo parecerá uma confusão sem sentido,

como mostra a figura 8.5.

Figura 8.5 – O programa calc.exe *do Windows aberto no Notepad.*

Como cada tipo de arquivo binário diferente deve ter tratado de uma maneira própria, este livro não abordará a leitura e a escrita direta de arquivos binários puros. Felizmente, muitos módulos facilitam trabalhar com arquivos binários – você explorará um deles, o módulo shelve, mais adiante neste capítulo.

Em Python, há três passos para ler e escrever em arquivos:

1. Chamar a função open() para que um objeto File seja retornado.
2. Chamar o método read() ou write() no objeto File.
3. Fechar o arquivo chamando o método close() no objeto File.

Abrindo arquivos com a função open()

Para abrir um arquivo com a função open(), passe um path em forma de string indicando o arquivo que você deseja abrir; poderá ser um path absoluto ou relativo. A função open() retorna um objeto File.

Teste isso criando um arquivo-texto chamado *hello.txt* usando o Notepad ou o TextEdit. Digite **Hello world!** como o conteúdo desse arquivo-texto e salve-o na pasta home de seu usuário. Em seguida, se você estiver usando Windows, digite o seguinte no shell interativo:

```
>>> helloFile = open('C:\\Users\\sua_pasta_home\\hello.txt')
```

Se estiver usando o OS X, digite o seguinte no shell interativo:

```
>>> helloFile = open('/Users/sua_pasta_home/hello.txt')
```

Não se esqueça de substituir *sua_pasta_home* pelo seu nome de usuário no computador. Por exemplo, meu nome de usuário é *asweigart*, portanto devo especificar 'C:\\Users\\asweigart\\hello.txt' no Windows.

Ambos os comandos abrirão o arquivo em modo de "leitura de texto simples" – ou em *modo de leitura* (read mode) para ser mais conciso. Quando um arquivo é aberto em modo de leitura, o Python permite somente ler dados do arquivo; você não poderá escrever no arquivo nem modificá-lo de forma alguma. O modo de leitura é o modo default para os arquivos que forem abertos em Python. No entanto, se não quiser contar com os defaults do Python, você poderá especificar explicitamente o modo passando o valor de string 'r' como o segundo argumento de open(). Desse modo, open('/Users/asweigart/hello.txt', 'r') e open('/Users/asweigart/hello.txt') fazem o mesmo.

A chamada a open() retorna um objeto File. Um objeto File representa um arquivo em seu computador; é somente outro tipo de valor em Python, muito semelhante às listas e aos dicionários com os quais você já tem familiaridade. No exemplo anterior, armazenamos o objeto File na variável helloFile. Agora, sempre que quisermos ler ou escrever no arquivo, poderemos fazer isso chamando os métodos do objeto File em helloFile.

Lendo o conteúdo dos arquivos

Agora que temos um objeto File, podemos começar a ler esse arquivo. Se quiser ler todo o conteúdo de um arquivo como um valor de string, utilize o método read() do objeto File. Vamos prosseguir com o objeto File para *hello.txt* que armazenamos em helloFile . Digite o seguinte no shell interativo:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

Se você pensar no conteúdo de um arquivo como um único valor de string extenso, o método read() retornará a string armazenada no arquivo.

De modo alternativo, podemos utilizar o método readlines() para obter uma *lista* de valores de string do arquivo, uma string para cada linha de texto. Por exemplo, crie um arquivo chamado *sonnet29.txt* no mesmo diretório em que está *hello.txt* e insira o texto a seguir nesse arquivo:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Não se esqueça de separar as quatro linhas com quebras de linha. Em seguida, digite o seguinte no shell interativo.

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my outcast
state,\n', And trouble deaf heaven with my bootless cries,\n', And look upon myself and
curse my fate,']
```

Observe que cada um dos valores de string termina com um caractere \n de quebra de linha, exceto a última linha do arquivo. Geralmente é mais fácil trabalhar com uma lista de strings do que com um único valor de string enorme.

Escrevendo em arquivos

O Python permite escrever conteúdo em um arquivo de modo muito semelhante à maneira como a função print() "escreve" strings na tela. Contudo não podemos escrever em um arquivo aberto em modo de leitura. Em vez disso, é necessário abri-lo em modo de "escrita de texto simples" ou de "adição de texto simples", ou seja, em *modo de escrita* (write mode) e em *modo de adição* (append mode), para sermos mais concisos.

O modo de escrita sobrescreverá o arquivo existente e começará do zero, como ocorre quando o valor de uma variável é sobrescrito com um novo valor. Passe 'w' como segundo argumento de open() para abrir o arquivo em modo de escrita. O modo de adição, por outro lado, adicionará o texto no final do arquivo existente. Podemos pensar nisso como a adição em uma lista que está em uma variável em vez de sobrescrever totalmente a variável. Passe

'a' como segundo argumento de open() para abrir o arquivo em modo de adição.

Se o nome do arquivo passado para open() não existir, tanto o modo de escrita quanto o modo de adição criarão um novo arquivo vazio. Após ler ou escrever em um arquivo, chame o método close() antes de abrir o arquivo novamente.

Vamos reunir todos esses conceitos. Digite o seguinte no shell interativo:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Inicialmente, abrimos *bacon.txt* em modo de escrita. Como ainda não há nenhum *bacon.txt*, o Python criará um. Chamar write() no arquivo aberto e passar o argumento de string 'Hello world! /n' a essa função fará a string ser escrita no arquivo e retornará o número de caracteres escritos, incluindo o caractere de quebra de linha. Em seguida, fechamos o arquivo.

Para adicionar texto ao conteúdo existente no arquivo em vez de substituir a string que acabamos de escrever, abrimos o arquivo em modo de adição. Escrevemos 'Bacon is not a vegetable.' no arquivo e o fechamos. Por fim, para exibir o conteúdo do arquivo na tela, nós o abrimos em seu modo de leitura default, chamamos read(), armazenamos o objeto File resultante em content, fechamos o arquivo e exibimos content.

Observe que o método write() não acrescenta um caractere de quebra de linha automaticamente no final da string, como é feito pela função print(). Você deverá acrescentar esse caractere por conta própria.

Salvando variáveis com o módulo shelve

Você pode salvar variáveis em seus programas Python em arquivos shelf binários usando o módulo shelve. Dessa maneira, seu programa poderá restaurar dados em variáveis que estão no disco rígido. O módulo shelve permitirá adicionar funcionalidades Save (Salvar) e Open (Abrir) em seu programa. Por exemplo, se você executar um programa e inserir alguns parâmetros de configuração, será possível salvar essas configurações em um arquivo shelf e, em seguida, fazer o programa carregá-las na próxima vez em que for executado.

Digite o seguinte no shell interativo:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

Para ler e escrever dados usando o módulo shelve, inicialmente é necessário importar esse módulo. Chame shelve.open() e passe um nome de arquivo; em seguida, armazene o valor de shelf retornado em uma variável. Você pode fazer alterações no valor de shelf como se fosse um dicionário. Quando terminar, chame close() no valor de shelf. Nesse caso, nosso valor de shelf está armazenado em shelfFile. Criamos uma lista cats e escrevemos shelfFile['cats'] = cats para armazenar a lista em shelfFile como um valor associado à chave 'cats' (como em um dicionário). Então chamamos close() em shelfFile.

Após executar o código anterior no Windows, você verá três novos arquivos no diretório de trabalho atual: *mydata.bak*, *mydata.dat* e *mydata.dir*. No OS X, somente um único arquivo *mydata.db* será criado.

Esses arquivos binários contêm os dados armazenados em seu shelf. O formato desses arquivos binários não é importante; você só precisa saber o que o módulo shelve faz, e não como ele faz. O módulo permite que você não se preocupe com o modo como os dados de seu programa são armazenados em um arquivo.

Seus programas poderão usar o módulo shelve para reabrir e obter posteriormente os dados desses arquivos shelf. Os valores de shelf não

precisam ser abertos em modo de leitura ou de escrita – ambas as operações serão permitidas após os valores serem abertos. Digite o seguinte no shell interativo:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Nesse caso, abrimos o arquivo shelf para verificar se nossos dados foram armazenados corretamente. Especificar shelfFile['cats'] retorna a mesma lista que armazenamos anteriormente, portanto sabemos que a lista está armazenada corretamente, e então chamamos close().

Assim como os dicionários, os valores de shelf têm métodos keys() e values() que retornarão as chaves e os valores do shelf em formatos semelhantes a listas. Como esses métodos retornam valores semelhantes a listas, e não listas de verdade, você deve passá-los à função list() para obtê-los em forma de lista. Digite o seguinte no shell interativo:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

O formato texto simples é útil para criar arquivos que serão lidos em um editor de texto como o Notepad ou o TextEdit, porém, se quiser salvar dados de seus programas Python, utilize o módulo shelve.

Salvando variáveis com a função pprint.pformat()

Lembre-se da seção “Apresentação elegante”, em que a função pprint.pprint() fazia uma apresentação elegante (“pretty print”) do conteúdo de uma lista ou de um dicionário na tela, enquanto a função pprint.pformat() retornava o mesmo texto na forma de uma string em vez de exibi-la. Essa string não só está

formatada para facilitar a leitura como também é um código Python sintaticamente correto. Suponha que você tenha um dicionário armazenado em uma variável e queira salvar essa variável e o seu conteúdo para usar futuramente. A chamada a pprint.pformat() fornecerá uma string que poderá ser gravada em um arquivo *.py*. Esse arquivo será seu próprio módulo, que poderá ser importado sempre que você quiser usar as variáveis armazenadas nele.

Por exemplo, digite o seguinte no shell interativo:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Nesse caso, importamos pprint para podermos usar pprint.pformat(). Temos uma lista de dicionários armazenada em uma variável cats. Para manter a lista em cats disponível mesmo após termos fechado o shell, utilizamos pprint.pformat() para retorná-la como uma string. Depois que tivermos os dados em cats na forma de uma string, será fácil gravar a string em um arquivo que chamaremos de *myCats.py*.

Os módulos que uma instrução import importa são apenas scripts Python. Quando a string de pprint.pformat() for salva em um arquivo *.py*, esse arquivo constituirá um módulo que poderá ser importado como qualquer outro.

Como os scripts Python em si são apenas arquivos-texto com a extensão de arquivo *.py*, seus programas Python podem até mesmo gerar outros programas Python. Esses arquivos podem ser então importados em scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

A vantagem de criar um arquivo *.py* (em oposição a salvar variáveis com o módulo shelve) está no fato de o conteúdo do arquivo poder ser lido e modificado por qualquer pessoa que utilize um editor de texto simples, por ele é um arquivo-texto. Na maioria das aplicações, porém, salvar dados usando o módulo shelve será a maneira preferida de salvar variáveis em um arquivo. Somente tipos de dados básicos como inteiros, números de ponto flutuante, strings, listas e dicionários podem ser gravados em um arquivo como texto simples. Objetos File, por exemplo, não podem ser codificados como texto.

Projeto: gerando arquivos aleatórios de provas

Suponha que você seja um professor de geografia, tenha 35 alunos em sua classe e queira aplicar uma prova surpresa sobre as capitais dos estados norte-americanos. Infelizmente, sua classe tem alguns alunos desonestos, e não é possível confiar neles acreditando que não vão colar. Você gostaria de deixar a ordem das perguntas aleatória para que cada prova seja única, fazendo com que seja impossível para alguém copiar as respostas de outra pessoa. É claro que fazer isso manualmente seria uma tarefa demorada e maçante. Felizmente, você conhece um pouco de Python.

Eis o que o programa faz:

- Cria 35 provas diferentes.
- Cria 50 perguntas de múltipla escolha para cada prova em ordem aleatória.
- Fornece a resposta correta e três respostas incorretas aleatórias para cada pergunta em ordem aleatória.
- Grava as provas em 35 arquivos-texto.
- Grava os gabaritos contendo as respostas em 35 arquivos-texto.

Isso significa que o código deverá fazer o seguinte:

- Armazenar os estados e suas capitais em um dicionário.
- Chamar open(), write() e close() para os arquivos-texto contendo as provas e os gabaritos com as respostas.
- Usar random.shuffle() para deixar a ordem das perguntas e as opções de múltipla escolha aleatórias.

Passo 1: Armazenar os dados da prova em um dicionário

O primeiro passo consiste em criar um esqueleto de script e preenchê-lo com os dados da prova. Crie um arquivo chamado *randomQuizGenerator.py* e faça com que ele tenha a seguinte aparência:

```
#! python3
# randomQuizGenerator.py – Cria provas com perguntas e respostas em
# ordem aleatória, juntamente com os gabaritos contendo as respostas.

u import random

# Os dados para as provas. As chaves são os estados e os valores são as capitais.
v capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee', 'Georgia':
'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois': 'Springfield',
'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas': 'Topeka', 'Kentucky':
'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine': 'Augusta', 'Maryland': 'Annapolis',
'Massachusetts': 'Boston', 'Michigan': 'Lansing', 'Minnesota': 'Saint Paul',
'Mississippi': 'Jackson', 'Missouri': 'Jefferson City', 'Montana': 'Helena', 'Nebraska':
'Lincoln', 'Nevada': 'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton',
'New Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh', 'North
Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City', 'Oregon':
'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence', 'South Carolina':
'Columbia', 'South Dakota': 'Pierre', 'Tennessee': 'Nashville', 'Texas': 'Austin',
'Utah': 'Salt Lake City', 'Vermont': 'Montpelier', 'Virginia': 'Richmond', 'Washington':
'Olympia', 'West Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Gera 35 arquivos contendo as provas.
w for quizNum in range(35):
    # TODO: Cria os arquivos com as provas e os gabaritos das respostas.

    # TODO: Escreve o cabeçalho da prova.

    # TODO: Embaralha a ordem dos estados.

    # TODO: Percorre todos os 50 estados em um loop, criando uma pergunta para cada
um.
```

Como esse programa ordenará as perguntas e as respostas de forma

aleatória, será necessário importar o módulo random u para que seja possível utilizar suas funções. A variável capitals v contém um dicionário com os estados norte-americanos como chaves e suas capitais como os valores. Como você quer criar 35 provas, o código que gera o arquivo com a prova e com o gabarito das respostas (marcado com comentários TODO por enquanto) estará em um loop for que executará 35 vezes w. (Esse número pode ser alterado para gerar qualquer quantidade de arquivos com as provas.)

Passo 2: Criar o arquivo com a prova e embaralhar a ordem das perguntas

Agora é hora de começar a preencher aqueles TODOs.

O código no loop será repetido 35 vezes – uma para cada prova –, portanto você deverá se preocupar somente com uma prova de cada vez no loop. Inicialmente, o arquivo propriamente dito para a prova será criado. Ele deve ter um nome único de arquivo e deve também ter algum tipo de cabeçalho-padrão, com espaços para o aluno preencher o nome, a data e o período da classe. Em seguida, será necessário obter uma lista dos estados em ordem aleatória, que poderá ser usada posteriormente para criar as perguntas e as respostas da prova.

Adicione as linhas de código a seguir em *randomQuizGenerator.py*:

```
#! python3
# randomQuizGenerator.py – Cria provas com perguntas e respostas em
# ordem aleatória, juntamente com os gabaritos contendo as respostas.

--trecho removido--

# Gera 35 arquivos contendo as provas.
for quizNum in range(35):
    # Cria os arquivos com as provas e os gabaritos das respostas.
u   quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
v   answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Escreve o cabeçalho da prova.
w   quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
```

```
    quizFile.write('\n\n')

    # Embaralha a ordem dos estados.
    states = list(capitals.keys())
x   random.shuffle(states)

    # TODO: Percorre todos os 50 estados em um loop, criando uma pergunta para cada
  um.
```

Os nomes dos arquivos para as provas serão *capitalsquiz<N>.txt*, em que *<N>* é um número único para a prova, proveniente de quizNum, que é o contador do loop for. O gabarito das respostas de *capitalsquiz<N>.txt* será armazenado em um arquivo-texto chamado *capitalsquiz_answers<N>.txt*. A cada passagem pelo loop, o placeholder %s em 'capitalsquiz%s.txt' e em 'capitalsquiz_answers%s.txt' será substituído por (quizNum + 1); sendo assim, a primeira prova e o primeiro gabarito criados serão *capitalsquiz1.txt* e *capitalsquiz_answers1.txt*. Esses arquivos serão criados por meio de chamadas à função open() em u e em v, com 'w' como segundo argumento para abri-los em modo de escrita.

As instruções write() em w criam um cabeçalho de prova para que o aluno possa preencher. Por fim, uma lista embaralhada dos estados norte-americanos é criada com a ajuda da função random.shuffle() x que reorganiza aleatoriamente os valores de qualquer lista recebida.

Passo 3: Criar as opções de resposta

Agora devemos gerar as opções de resposta para cada pergunta, que corresponderão às múltiplas escolhas de A a D. Será necessário criar outro loop for – esse servirá para gerar o conteúdo de cada uma das 50 perguntas da prova. Em seguida, haverá um terceiro loop for aninhado para gerar as opções de múltipla escolha para cada pergunta. Faça seu código ter o seguinte aspecto:

```
#! python3
# randomQuizGenerator.py – Cria provas com perguntas e respostas em
# ordem aleatória, juntamente com os gabaritos contendo as respostas.

--trecho removido--
```

```
    # Percorre todos os 50 estados em um loop, criando uma pergunta para cada um.
    for questionNum in range(50):

        # Obtém respostas corretas e incorretas.
u       correctAnswer = capitals[states[questionNum]]
v       wrongAnswers = list(capitals.values())
w       del wrongAnswers[wrongAnswers.index(correctAnswer)]
x       wrongAnswers = random.sample(wrongAnswers, 3)
y       answerOptions = wrongAnswers + [correctAnswer]
z       random.shuffle(answerOptions)

        # TODO: Grava a pergunta e as opções de resposta no arquivo de prova.

        # TODO: Grava o gabarito com as respostas em um arquivo.
```

A resposta correta é fácil de ser obtida – ela está armazenada como um valor no dicionário `capitals` u. Esse loop percorrerá os estados na lista `states` embaralhada, de `states[0]` a `states[49]`, encontrará cada estado em `capitals` e armazenará a capital correspondente a esse estado em `correctAnswer`.

A lista de possíveis respostas incorretas é mais complicada. Podemos obtê-la duplicando *todos* os valores do dicionário `capitals` v, apagando a resposta correta w e selecionando três valores aleatórios dessa lista x. A função `random.sample()` facilita fazer essa seleção. Seu primeiro argumento é a lista em que você deseja fazer a seleção; o segundo argumento é a quantidade de valores que você quer selecionar. A lista completa de opções de resposta será a combinação dessas três respostas incorretas com a resposta correta y. Por fim, as respostas devem ser embaralhadas z para que a resposta correta não seja sempre a opção D.

Passo 4: Gravar conteúdo nos arquivos de prova e de respostas

Tudo que resta fazer é gravar a pergunta no arquivo de prova e a resposta no arquivo com o gabarito das respostas. Faça seu código ter o seguinte aspecto:

```
#! python3
# randomQuizGenerator.py – Cria provas com perguntas e respostas em
```

```
    # ordem aleatória, juntamente com os gabaritos contendo as respostas.

    --trecho removido--

        # Percorre todos os 50 estados em um loop, criando uma pergunta para cada um.
        for questionNum in range(50):
            --trecho removido--

            # Grava a pergunta e as opções de resposta no arquivo de prova.
            quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
                states[questionNum]))
u           for i in range(4):
v               quizFile.write('    %s. %s\n' % ('ABCD'[i], answerOptions[i]))
            quizFile.write('\n')

            # Grava o gabarito com as respostas em um arquivo.
w           answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[
                answerOptions.index(correctAnswer)]))
        quizFile.close()
        answerKeyFile.close()
```

Um loop `for` que percorre os inteiros de `0` a `3` escreverá as opções de resposta da lista `answerOptions` u. A expressão `'ABCD'[i]` em v trata a string `'ABCD'` como um array e será avaliada como `'A'`,`'B'`, `'C'` e então `'D'` a cada respectiva iteração pelo loop.

Na última linha w, a expressão `answerOptions.index(correctAnswer)` encontrará o índice inteiro da resposta correta nas opções de resposta ordenadas aleatoriamente e `'ABCD'[answerOptions.index(correctAnswer)]` será avaliada com a letra da resposta correta, que será gravada no arquivo contendo o gabarito.

Após executar o programa, seu arquivo *capitalsquiz1.txt* terá a aparência a seguir, embora, é claro, suas perguntas e as opções de resposta possam ser diferentes daquelas mostradas aqui, pois dependem do resultado de suas chamadas a `random.shuffle()`:

```
Name:

Date:

Period:
```

State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
   A. Hartford
   B. Santa Fe
   C. Harrisburg
   D. Charleston

2. What is the capital of Colorado?
   A. Raleigh
   B. Harrisburg
   C. Denver
   D. Lincoln

--trecho removido--

O arquivo-texto *capitalsquiz_answers1.txt* correspondente terá o seguinte aspecto:

1. D
2. C
3. A
4. C

--trecho removido--

Projeto: Multiclipboard

Suponha que você tenha a tarefa maçante de preencher diversos formulários em uma página web ou em um software com vários campos de texto. O clipboard evita que seja necessário digitar o mesmo texto repetidamente. Porém somente um texto pode estar no clipboard a cada instante. Se você tiver diversas porções de texto diferentes que devam ser copiadas e coladas, será necessário marcar e copiar os mesmos dados repetidamente.

Podemos criar um programa Python que controle diversas porções de texto. Esse "multiclipboard" se chamará *mcb.pyw* (pois "mcb" é mais conciso para digitar do que "multiclipboard"). A extensão *.pyw* quer dizer que o Python não mostrará uma janela do Terminal quando executar esse programa. (Consulte o apêndice B para obter mais detalhes.)

O programa salvará cada porção de texto do clipboard com uma palavra-chave. Por exemplo, ao executar py mcb.pyw save spam, o conteúdo atual do clipboard será salvo com a palavra-chave *spam*. Esse texto poderá ser posteriormente carregado para o clipboard novamente se py mcb.pyw spam for executado. Se o usuário se esquecer das palavras-chave existentes, ele poderá executar py mcb.pyw list para que uma lista de todas as palavras-chaves seja copiada para o clipboard.

Eis o que o programa faz:

- O argumento de linha de comando para a palavra-chave é verificado.
- Se o argumento for save, o conteúdo do clipboard será salvo com a palavra-chave.
- Se o argumento for list, todas as palavras-chaves serão copiadas para o clipboard.
- Caso contrário, o texto da palavra-chave será copiado para o clipboard.

Isso significa que o código deverá fazer o seguinte:

- Ler os argumentos de linha de comando em sys.argv.
- Ler e escrever no clipboard.
- Salvar e carregar um arquivo shelf.

Se estiver usando Windows, você poderá executar facilmente esse script a partir da janela Run... (Executar) criando um arquivo batch chamado *mcb.bat* com o conteúdo a seguir:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Passo 1: Comentários e configuração do shelf

Vamos começar criando o esqueleto de um script com alguns comentários e uma configuração básica. Faça seu código ter o seguinte aspecto:

```
#! python3
# mcb.pyw – Salva e carrega porções de texto no clipboard.
u # Usage: py.exe mcb.pyw save <palavra-chave> – Salva clipboard na palavra-chave.
#       py.exe mcb.pyw <palavra-chave> - Carrega palavra-chave no clipboard.
#       py.exe mcb.pyw list – Carrega todas as palavras-chave no clipboard.

v import shelve, pyperclip, sys
```

```
w mcbShelf = shelve.open('mcb')

  # TODO: Salva conteúdo do clipboard.

  # TODO: Lista palavras-chave e carrega conteúdo.

  mcbShelf.close()
```

Colocar informações gerais de uso em comentários no início do arquivo u é uma prática comum. Se algum dia se esquecer de como seu script deve ser executado, você sempre poderá dar uma olhada nesses comentários para recordar. Em seguida, importe seus módulos v. Copiar e colar exigirá o módulo pyperclip, enquanto ler os argumentos de linha de comando exigirá o módulo sys. O módulo shelve também será útil: sempre que o usuário quiser salvar uma nova porção de texto do clipboard, você poderá salvá-la em um arquivo shelf. Então, quando o usuário quiser copiar o texto de volta ao clipboard, você abrirá o arquivo shelf e carregará o texto de volta em seu programa. O nome do arquivo shelf terá o prefixo *mcb* w.

Passo 2: Salvar o conteúdo do clipboard com uma palavra-chave

O programa realiza tarefas diferentes conforme o usuário queira salvar texto em uma palavra-chave, carregar texto no clipboard ou listar todas as palavras-chave existentes. Vamos tratar o primeiro caso. Faça seu código ter o seguinte aspecto:

```
  #! python3
  # mcb.pyw – Salva e carrega porções de texto no clipboard.
  --trecho removido--

  # Salva conteúdo do clipboard.
u if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
v       mcbShelf[sys.argv[2]] = pyperclip.paste()
  elif len(sys.argv) == 2:
w     # TODO: Lista palavras-chave e carrega conteúdo.
  mcbShelf.close()
```

Se o primeiro argumento da linha de comando (que sempre estará no índice 1 da lista sys.argv) for 'save' u, o segundo argumento da linha de comando será a palavra-chave para o conteúdo atual do clipboard. A palavra-chave será usada como chave para mcbShelf e o valor será o texto que está no clipboard no momento v.

Se houver apenas um argumento na linha de comando, suporemos que será 'list' ou uma palavra-chave para carregar o conteúdo no clipboard. Esse código será implementado mais adiante. Por enquanto, basta colocar um comentário TODO nesse ponto w.

Passo 3: Listar palavras-chaves e carregar o conteúdo de uma palavra-chave

Por fim, vamos implementar os dois casos restantes: o usuário quer carregar texto no clipboard a partir de uma palavra-chave ou quer ver uma lista de todas as palavras-chaves disponíveis. Faça seu código ter o seguinte aspecto:

```
  #! python3
  # mcb.pyw – Salva e carrega porções de texto no clipboard.
  --trecho removido--

  # Salva conteúdo do clipboard.
  if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
        mcbShelf[sys.argv[2]] = pyperclip.paste()
  elif len(sys.argv) == 2:
     # Lista palavras-chave e carrega conteúdo.
u    if sys.argv[1].lower() == 'list':
v       pyperclip.copy(str(list(mcbShelf.keys())))
     elif sys.argv[1] in mcbShelf:
w       pyperclip.copy(mcbShelf[sys.argv[1]])
  mcbShelf.close()
```

Se houver somente um argumento de linha de comando, inicialmente vamos verificar se é 'list' u. Em caso afirmativo, uma representação em string da lista de chaves do shelf será copiada para o clipboard v. O usuário poderá colar essa lista em um editor de texto que estiver aberto e lê-la.

Caso contrário, podemos supor que o argumento de linha de comando é uma palavra-chave. Se essa palavra-chave existir no shelf mcbShelf como

chave, poderemos carregar o valor no clipboard w.

É isso! Iniciar esse programa envolve passos diferentes conforme o sistema operacional utilizado pelo seu computador. Consulte o apêndice B para ver os detalhes em seu sistema operacional.

Lembre-se do programa de repositório de senhas criado no capítulo 6 que armazenava as senhas em um dicionário. Atualizar as senhas exigia alterar o código-fonte do programa. Isso não é ideal, pois os usuários comuns não se sentem à vontade alterando um código-fonte para atualizar seus softwares. Além do mais, sempre que modificar o código-fonte de um programa, você correrá o risco de introduzir novos bugs acidentalmente. Ao armazenar os dados de um programa em um local diferente do lugar em que está o código, você poderá deixar seus programas mais fáceis para outras pessoas usarem e mais resistentes a bugs.

Resumo

Os arquivos estão organizados em pastas (também chamadas de diretórios) e um path descreve a localização de um arquivo. Todo programa que estiver executando em seu computador tem um diretório de trabalho atual, que permite especificar os paths de arquivo em relação à localização atual em vez de sempre exigir a digitação do path completo (ou absoluto). O módulo `os.path` contém muitas funções para manipular paths de arquivo.

Seus programas também poderão interagir diretamente com o conteúdo de arquivos-texto. A função `open()` pode abrir esses arquivos para ler seus conteúdos na forma de uma string longa (com o método `read()`) ou como uma lista de strings (com o método `readlines()`). A função `open()` pode abrir arquivos em modo de escrita ou de adição para criar novos arquivos-texto ou para adicionar dados em um arquivo-texto existente, respectivamente.

Nos capítulos anteriores, usamos o clipboard como uma maneira de inserir grandes quantidades de texto em um programa em vez de digitar tudo. Agora você pode fazer seus programas lerem arquivos diretamente do disco rígido, o que é uma melhoria significativa, pois os arquivos são muito menos voláteis que o clipboard.

No próximo capítulo, aprenderemos a lidar com os arquivos propriamente

ditos, copiando, apagando, renomeando, movendo e realizando outras operações com esses arquivos.

Exercícios práticos

1. Um path relativo é relativo a quê?
2. Um path absoluto começa com qual informação?
3. O que as funções os.getcwd() e os.chdir() fazem?
4. O que são as pastas . e .. ?
5. Em *C:\bacon\eggs\spam.txt*, qual parte corresponde ao nome do diretório e qual parte é o nome base?
6. Quais são os três argumentos de "modo" que podem ser passados para a função open()?
7. O que acontecerá se um arquivo existente for aberto em modo de escrita?

8 Qual é a diferença entre os métodos read() e readlines()?

9. Que estrutura de dados um valor de shelf lembra?

Projetos práticos

Para exercitar, crie o design e implemente os programas a seguir.

Estendendo o multiclipboard

Estenda o programa multiclipboard deste capítulo para que ele tenha um argumento de linha de comando delete <*palavra-chave*> que apagará uma palavra-chave do shelf. Em seguida, acrescente um argumento de linha de comando delete que apagará *todas* as palavras-chaves.

Mad Libs

Crie um programa Mad Libs que leia arquivos-texto e permita que o usuário acrescente seus próprios textos em qualquer local em que a palavra *ADJECTIVE*, *NOUN*, *ADVERB* ou *VERB* aparecer no arquivo-texto. Por exemplo, um arquivo-texto poderá ter o seguinte aspecto:

The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was unaffected by these events.

O programa deve localizar essas ocorrências e pedir que o usuário as substitua.

Enter an adjective:
silly
Enter a noun:
chandelier
Enter a verb:
screamed
Enter a noun:
pickup truck

O texto a seguir deverá ser criado:

The silly panda walked to the chandelier and then screamed. A nearby pickup truck was unaffected by these events.

O resultado deverá ser exibido na tela e salvado em um novo arquivo-texto.

Pesquisa com regex

Crie um programa que abra todos os arquivos *.txt* de uma pasta e procure todas as linhas que correspondam a uma expressão regular fornecida pelo usuário. O resultado deverá ser exibido na tela.