Capítulo 10

Trabalhando com arquivos

Neste capítulo, aprenderemos a manipular arquivos, para então podermos ler os arquivos de dados da base que estamos usando para popular nosso modelo de objetos, desenvolvido no capítulo 7.

10.1 Importando dados para as tabelas

Até o momento, criamos algumas classes que serão a base do nosso aplicativo, mas elas ainda não estão nos permitindo muitas coisas. O que queremos agora é começar a ler os arquivos de dados e a criar as instâncias dessas tabelas, usando os dados das bases das quais podemos fazer download. Depois que as tabelas foram criadas e os dados dos arquivos lidos, o objetivo é usá-las para realizar consultas nos dados.

Para ler arquivos, temos a função *builtin* (embutida), chamada `open()`, que abre um arquivo e retorna um objeto do tipo arquivo. No Python 3, exis-

tem três tipos de arquivos: binários, binários bufferizados e de texto. O arquivo que vamos ler é um arquivo texto. Vamos começar com o exemplo mais simples possível e aproveitar para ver que tipo tem um objeto arquivo quando aberto com os parâmetros padrão:

```
>>> data = open('data/data/ExecucaoFinanceira.csv', 'r')
>>> data
<_io.TextIOWrapper name='ExecucaoFinanceira.csv'
mode='r' encoding='UTF-8'>
>>> data.close()
```

O código anterior passa como parâmetro o caminho e `'r'` que é o modo, no caso `read` ou leitura, que serve para abrir o arquivo para leitura dos dados. Quando mencionamos objeto do tipo arquivo, a ideia é que, independente de seu tipo, sua interface seja igual para todos. Mesmo que Python não tenha interfaces como Java, os objetos ainda podem seguir determinadas padronizações de métodos, ou até mesmo herdar de classes abstratas.

A assinatura completa da função é: `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`. Os modos informam se o arquivo é texto ou binário, e para quais operações será aberto (leitura, escrita etc.).

Veja a lista de modos a seguir:

```
'r'    open for reading (default)
'w'    open for writing, truncating the file first
'x'    open for exclusive creation, failing if the file already
exists
'a'    open for writing, appending to the end of the file if it
exists
'b'    binary mode
't'    text mode (default)
'+'    open a disk file for updating (reading and writing)
```

Vale lembrar que os modos podem ser combinados, se pertinente, como em: `wb`, sendo escrita binária (*write binary*); ou `rt`, como leitura texto (*read text*).

O parâmetro `buffering` permite algumas opções: `-1` é o valor padrão, que significa usar as configurações do sistema; `0` para não usar buffer (so-

mente no modo binário); `1` para usar um buffer por linhas (válido somente em arquivo texto); e, em valores > 1, o tamanho do buffer passa a ser o valor passado como parâmetro.

O parâmetro `encoding` pode ser a string com o nome do formato, como `"utf-8"`.

> Os outros parâmetros são muito específicos e podem ser consultados na documentação oficial, em https://docs.python.org/3/library/functions.html#open.

Voltando ao exemplo, veja que o objeto retornado é do tipo `TextIOWrapper`. Essa classe é uma implementação de arquivos de texto, que, além de ter as funções comuns a todo tipo de objeto do tipo arquivo, tem também funções específicas que só podem ser usadas para arquivos abertos em modo texto.

Os tipos de arquivo de texto trabalham com strings em vez de bytes, como nos arquivos binários. No capítulo 2, vimos que no Python 3 todas strings são **unicode** e, portanto, quando escritas em arquivo ou enviadas pela rede, devem ser convertidas para bytes através de um *encoder*. Nos tipos de arquivo de texto, mecanismos internos já fazem a conversão de strings para bytes (e vice-versa).

Como queremos olhar para os tipos de arquivo de forma mais genérica, vamos nos concentrar nos métodos em comum a todos os tipos de arquivo, e ver a seguir dois dos mais importantes: `read()` e `write()`.

10.2 Lendo arquivos

A função para ler dados é `read()`. Quando a invocamos sem parâmetros, todo o conteúdo do arquivo é lido e retornado. Se o arquivo for muito grande, o seu programa pode consumir muita memória. Uma alternativa é especificar a quantidade de bytes a ser lida, como `read(4096)`. Vamos ver o exemplo:

```
>>> data = open('ExecucaoFinanceira.csv', 'r')
>>> data.read(19)
```

```
'1;2;132;CONSTRUTORA'
>>> data.close()
```

O código exibiu os 19 primeiros bytes do arquivo de execuções financeiras. Como estamos trabalhando com arquivos de texto que seguem o formato CSV, geralmente as quebras de linha `"\n"` delimitam o final de uma informação. Por exemplo, no caso de um arquivo de dados `copa_transparente` [1], cada linha significa um registro naquela tabela. Logo, uma linha no arquivo `ExecucaoFinanceira.csv` contém a informação de uma Execução Financeira.

Existe uma forma melhor de ler, linha a linha, um arquivo de texto, que veremos na sequência.

Iterando nas linhas

No nosso aplicativo, como todos os arquivos têm menos de 10Mb, podemos lê-los de uma vez só, apenas chamando `read()`, sem parâmetro algum. Em nosso caso, estamos lidando com um CSV, então queremos ler linha a linha. Em Python, podemos usar o método `readline()` ou iterar as linhas do arquivo, usando o comando `for` no próprio objeto do tipo arquivo. Veja o exemplo:

```
data = open('data/data/ExecucaoFinanceira.csv', 'r')
for line in data:
    print(line)
data.close()
```

O método `readline()` está disponível apenas para arquivos de texto. A iteração usando o comando `for` também funciona em arquivos binários, mas continuará fazendo a quebra pelo caractere `"\n"`.

Se quisermos todas as linhas em uma lista de strings, em que cada item da lista é uma linha do arquivo, podemos usar a função `readlines()`. A desvantagem é que, novamente, em arquivos grandes, pode ser criada uma lista muito grande em memória. Deve-se analisar caso a caso se é necessário ter tudo em memória simultaneamente, ou se ter acesso apenas a uma linha por vez já atende à necessidade.

Para ver um exemplo da função `readlines()`, vamos ver mais um tipo de arquivo de texto que é muito comum: `StringIO`. Um `StringIO` é um *stream* de texto em memória, que implementa o contrato da API de manipulação de arquivos de texto. É muito útil quando queremos usar um código que espera um objeto do tipo arquivo e nós podemos passar um objeto compatível, mas que por dentro tem uma string que é, na verdade, o conteúdo do arquivo.

```
>>> import io
>>> data = io.StringIO("a\nb\nc")
>>> lines = data.readlines()
>>> print(lines)
['a\n', 'b\n', 'c']
>>> data.close()
>>> data.closed
True
```

Para o código, `data` é um arquivo e pode ser usado como arquivo em códigos que esperam um tipo arquivo texto.

Como esperado, a função `readlines` retornou uma lista com cada linha do arquivo. Repare que as linhas que têm o caractere `"\n"` vêm com ele na string, e as linhas que não têm (no caso, apenas a última), vêm sem.

> **Iterar *versus* ler todas as linhas**
>
> A vantagem de iterar nas linhas é que em nenhum momento precisamos ter em memória um espaço grande o suficiente para caber todo o conteúdo. Para arquivos muito grandes, isso pode fazer uma boa diferença.
>
> Em alguma situações, pode ser desejável ter o conteúdo todo em memória. Então, cada situação deve ser avaliada separadamente.

10.3 Fechando arquivos

Programas "bem comportados" fecham os arquivos abertos ao final de sua execução. Nos nossos exemplos, fizemos isso usando o método `close()`.

Todos os tipos de arquivo implementam o método `close()`, até mesmo `StringIO`, que não tem um arquivo real aberto e que, depois do método `close()` chamado, seu conteúdo não pode ser mais lido. Nesses casos, a chamada ao método `read()`, em um arquivo fechado, gera um `ValueError`.

Podemos verificar se um objeto do tipo arquivo está fechado usando a propriedade `closed`. Veja o exemplo que ilustra o que falamos:

```
>>> import io
>>> data = io.StringIO("a\nb\nc")
>>> lines = data.readlines()
>>> print(lines)
['a\n', 'b\n', 'c']
>>> data.close()
>>> data.closed
True
>>> data.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

10.4 Abrindo arquivos com o comando with

Em Python, existe uma forma **pythônica** de se trabalhar com arquivos usando o comando `with`. Esse comando introduzido pela PEP-343 [4] tem o objetivo de simplificar o uso repetido de comandos `try/finally` em algumas situações. Conceitualmente, temos um gerenciador de contexto, que é uma classe de cujo ciclo de vida dois métodos fazem. Por meio desses métodos, conseguimos executar código em momentos como entrada e saída de um bloco com `with`.

Com esse comando, não precisamos nos preocupar em fechar explicitamente o arquivo, já que o próprio gerenciador de contexto será chamado no final do bloco `with`, e chamará o método `close()` do arquivo em questão. Vamos usá-lo no nosso primeiro exemplo:

```
with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    content = data.read()
    print(content)
```

Nesse exemplo, o objeto retornado pela função `open()` é compatível com o protocolo de gerenciador de contexto. Portanto, quando o bloco `with` termina, o arquivo retornado por `open('ExecucaoFinanceira.csv', 'r')` é fechado. Algumas bibliotecas de acesso a banco de dados usam esse comando para delimitar conexões abertas, ou até mesmo transações em bancos de dados.

Se juntarmos o comando `with` com leitura de arquivo linha a linha e *split* de strings, já conseguimos obter todos os valores de execuções financeiras da base de dados da copa. Veja o exemplo:

```
with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    for line in data:
        print(line.split(';')[5])
```

Veremos na saída valores de execuções financeiras até que um erro seja exibido quando uma linha mal formada for encontrada. Por enquanto isso não é um problema já que o foco é sobre abrir um arquivo, ler linha a linha e deixar que o próprio ambiente se encarregue de fecha-lo ao final do uso.

10.5 Escrevendo em arquivos

No nosso primeiro programa, no capítulo 4, vimos um exemplo de escrita em arquivo no código que faz o download do arquivo da base de dados. A escrita no arquivo se deu basicamente em duas linhas do código. Vamos analisar:

```
>>> out_file = io.FileIO(file_path, mode="w")
```

Veja o código anterior usando a função `open()`:

```
>>> out_file = open(file_path, mode="wb") # modo write binary
```

Agora, as chamadas da função `write()`:

```
>>> out_file.write(bytes_to_write)
```

Por se tratar de um arquivo `ZIP` – que tem conteúdo binário –, usamos a classe `FileIO` que trabalha com bytes em vez de strings. Porém, poderíamos usar a função `open()`, sinalizando arquivo binário, exatamente como no

exemplo anterior. Além disso, abrimos o arquivo com modo `w`, que é o modo de escrita (*write*), e que permite que a função `write()` seja chamada. O código completo não faz nada além de ler dados da resposta do servidor e escrever em um arquivo local.

Se estivéssemos trabalhando com arquivos texto, a única diferença seria o método `write()`, que estaria esperando uma string (e não bytes), como no nosso exemplo. É muito importante escolher o tipo correto dependendo do arquivo que será lido, pois, caso contrário, as coisas podem não funcionar como esperado.

10.6 Navegação avançada com seek()

Uma função base de arquivos é a `seek()`. Essa função é muito semelhante à `fseek()` da linguagem C. Com ela, você pode colocar o ponteiro do arquivo no lugar que desejar. Isso é muito útil quando queremos ler o mesmo trecho de um arquivo diversas vezes.

Veja o exemplo:

```
>>> data = open('ExecucaoFinanceira.csv', 'r')
>>> data.read(5)
'1;2;1'
>>> data.read(5)
'32;CO'
>>> data.read(5)
'NSTRU'
>>> data.read(5)
'TORA '
>>> data.seek(0)
0
>>> data.read(20)
'1;2;132;CONSTRUTORA '
```

Repare que, a cada chamada de `read()`, o ponteiro do arquivo é atualizado e a nova leitura é iniciada em uma outra região. Por meio do uso da função `fseek()`, foi possível retornar ao início do arquivo para realizar a leitura.

10.7 Conclusão

Neste capítulo, vimos como abrir e fechar arquivos, e como usar o comando `with` para que eles sejam fechados automaticamente. Depois aprendemos algumas formas de se obter os dados dos arquivos, seja lendo uma quantidade específica de bytes, iterando linha a linha com o loop `for`, ou até mesmo pegando uma lista com todas as linhas do arquivo.

Além disso, vimos também como usar os modos de abertura, interferindo nos objetos retornados, e quais operações podemos realizar neles. Por fim, aprendemos como escrever em arquivos e em quais detalhes devemos ficar atentos. A seguir, veremos mais recursos disponíveis na biblioteca padrão, mas que não estão embutidos na linguagem e precisamos importá-los.