10

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

10

Manipulação de Arquivos

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo veremos o que é a persistência de dados em arquivos e como utilizá-la em Python. Vamos aprender a criar, abrir, escrever e consultar dados em arquivos de texto e arquivos binários através de programas escritos em Python.

10.1. Agradecimentos

Gostaria de agradecer ao Professor Me. Lucio Nunes de Lira pela contribuição dada à elaboração deste capítulo.

10.2. Introdução

Até este ponto da disciplina, nossos programas já utilizaram muitos recursos e estruturas úteis para receber dados, processá-los e gerar saídas. Porém, ao desligar o computador ou simplesmente fechar o programa, essas saídas são perdidas! As entradas também são descartadas, e caso seja necessário processá-las novamente teremos que digitá-las, o que pode ser um problema quando a entrada é composta por muitos dados.

Então, o que queremos é preservar as saídas em um espaço de memória não volátil, ou seja, queremos persistir os dados em outro local que não seja a memória RAM, que é onde estão as variáveis e os códigos dos programas enquanto eles são executados. Exemplos de memórias não voláteis são HDD, SSD, Pen Drive, SD Card ou mesmo em servidores na nuvem.

Como mencionado, também é útil conseguir as entradas para os programas de alguma fonte que não seja um teclado, pois a necessidade de digitar valores algumas vezes pode ser impraticável, dependendo do tamanho da entrada e da frequência com que o programa precise lê-las.

Neste documento veremos como fazer isso em Python, com o uso de arquivos tanto para persistir os dados (guardando as saídas do programa), quanto para facilitar a inserção de dados (lendo as entradas do programa).

10.3. Arquivos

De maneira simplificada, um arquivo é uma área de memória onde podemos realizar a leitura e a escrita de dados. Essa área geralmente está localizada em um dispositivo que permite a persistência dos dados, que é justamente o que não ocorre na memória RAM. A persistência consiste em garantir que, mesmo ao cortar o fluxo de energia do computador ou fechar o programa, os dados permanecerão guardados e poderão ser acessados no futuro.

Como o gerenciamento da área de memória onde serão guardados os dados do arquivo é de responsabilidade do sistema operacional, não é necessário que o programador se encarregue desse nível de detalhes para criar programas que usem arquivos, pelo menos não em Python. Porém, existem procedimentos básicos que devemos conhecer para trabalhar com arquivos. Por exemplo, é necessário saber o local (caminho) onde o arquivo está para ser consultado (leitura) e definir o nome e o local onde será guardado ou alterado (escrita).

Existem arquivos com diversas finalidades. Em seu smartphone provavelmente há fotos, vídeos, músicas e textos. Eles são arquivos! Porém de tipos diferentes e por isso precisam de programas diferentes para serem abertos e lidos, afinal cada tipo de arquivo possui peculiaridades que precisam ser consideradas, de modo a serem corretamente interpretados pelo computador. Ainda no exemplo de um smartphone, em geral os aplicativos usam um banco de dados em arquivo, como por exemplo SQLite, que é uma base de dados relacional salva em um único arquivo e que não precisa de nenhum tipo de servidor, e é usada por padrão tanto no Android quanto no iOS. Outro exemplo é a Couchbase Lite, uma base de dados não relacional (NoSQL) que guarda os dados em arquivos JSON e possui APIs[1] nativas tanto para Android quanto para iOS.

10.3.1. Abertura e criação de arquivos

Para trabalhar com arquivos em Python, a primeira coisa que precisamos fazer é abrir o arquivo, o que é feito com a função integrada `open`. Esta função recebe um parâmetro obrigatório, que é o caminho para o arquivo e pode receber diversos parâmetros opcionais que alteram a forma como o arquivo será aberto e tratado pelo Python.

[1] API é um acrônimo para "*Application Programming Interface*", que em português é "Interface de Programação de Aplicação", e podemos entendê-la como uma interface que expõe métodos e funções de uma aplicação para serem utilizados por outras aplicações, sem que elas precisem (ou possam) acessar os detalhes de implementação da aplicação em questão. Uma API pode ser feita tanto para aplicações web quanto para programas locais, que rodam no próprio dispositivo.

Em Python há duas formas diferentes de se abrir um arquivo: como um binário, em que o arquivo é lido e retornado como uma sequência de bytes sem aplicar nenhum tipo de decodificação, ou então como arquivo de texto, em que o arquivo é lido e decodificado usando a codificação padrão da plataforma, ou a codificação passada nos argumentos, caso isso seja feito, e seu conteúdo é então convertido para uma *string*. Além disso, podemos definir as permissões que iremos dar ao Python para manipular o arquivo: somente leitura, somente escrita ou ambas.

Para definirmos o modo como um arquivo será aberto, devemos passar um segundo parâmetro, além do caminho para o arquivo, conforme mostra a Tabela 10.1 (PSF, 2021a).

Tabela 10.1: Modos de abertura de um arquivo em Python. Fonte: PSF, 2021a

Caractere	Significado
`'r'`	abre para leitura (padrão)
`'w'`	abre para escrita, truncando[2] o arquivo primeiro caso ele exista
`'x'`	abre para criação exclusiva, falhando caso o arquivo exista
`'a'`	abre para escrita, anexando ao final do arquivo caso ele exista
`'b'`	modo binário
`'t'`	modo texto (padrão)
`'+'`	abre para atualização (leitura e escrita)

Fonte: do autor, 2021

Observações a partir da Tabela 10.1:

- O modo padrão é `'r'`, que é o mesmo que `'rt'`;
- Os modos `'w+'` e `'w+b'` abrem o arquivo para leitura e escrita, truncando-o primeiro (o conteúdo original será sobrescrito).
- Os modos `'r+'` e `'r+b'` abrem o arquivo para leitura e escrita sem truncá-lo.

É importante lembrar que quando abrimos um arquivo, ele fica bloqueado pelo processo que o está utilizando, então após realizar as operações necessárias de leitura/escrita de dados, é imprescindível fecharmos o arquivo. Para isso usamos o método `close` do objeto de arquivo que foi gerado pela função `open`. Veja o exemplo na Codificação 10.1.

[2] Truncar o arquivo significa que todo seu conteúdo inicial será apagado e o arquivo será efetivamente sobrescrito neste modo de abertura.

Codificação 10.1: Exemplo de abertura, processamento e fechamento de um arquivo

```
f = open('teste.txt', 'w')
f.write('Olá Mundo!\n')
f.close()
```

Fonte: do autor, 2021

Após executar o código da Codificação 10.1, será criado um arquivo com o nome "texto.txt" na mesma pasta do arquivo Python executado, contendo o texto "Olá Mundo!". Faça o teste no seu computador e veja o resultado.

10.3.1.1. Considerações sobre o caminho de arquivos

Nos exemplos da seção 10.3.1 o arquivo está na mesma pasta, então basta colocar o nome do arquivo, incluindo a extensão. Mas muitas vezes esse não é o caso, e o arquivo pode estar em outra pasta, de modo que podemos passar o caminho para esta pasta de duas formas diferentes:

- Caminho absoluto: quando o caminho começa a partir da unidade básica do sistema, como por exemplo `'C:\Users\Public\teste.txt'` no windows ou `'/home/myuser/documents/teste.txt'` no Linux.
- Caminho relativo: podemos passar o caminho para o arquivo a partir da localização do arquivo atual, usando dois pontos finais para indicar ao Python para buscar uma pasta acima, como por exemplo `'..\..\teste.txt'` irá buscar por um arquivo chamado `teste.txt` que se encontra dois níveis acima na hierarquia de diretórios (no Linux basta trocar a `'\'` por `'/'`).

No entanto, a forma que o Windows define os caminhos conflita com a forma que o Python define caracteres especiais nas *strings*, pois em Python, o caractere `'\'` possui um significado especial, que é o caractere de escape.

Por exemplo, se escrevemos em Python o seguinte caminho `'C:\nova-pasta\teste.txt'`, o `'\'` altera o significado do caractere imediatamente após ele, no caso `'n'` e `'t'`, que deixam de representar as letras e passam a ser interpretados, respectivamente, como uma quebra de linha e um caractere de tabulação (tab), de modo que não será possível acessar o caminho desejado.

Para resolver essa questão, há duas formas: podemos escapar o caractere `'\'`, o que é feito com a própria `'\'`: `'\\'`. A primeira `'\'` está alterando o significado da segunda, que deixa de ser um caractere especial e passa a representar o próprio caractere em si. Portanto, o caminho deve ser escrito como: `'C:\\nova-pasta\\teste.txt'`.

Há ainda outra forma de resolver este problema, que foi introduzida na versão 3.1 do Python, que é usar uma *string* "crua", isto é, uma *string* na qual o caractere `'\'` não possui nenhum significado especial, e portanto não precisa ser escapado. Essa string

é construída prefixando-a com a letra `r`, que vem do inglês "*raw string*", e o caminho para o arquivo fica então: `r'C:\nova-pasta\teste.txt'`.

Caso seja necessário usar uma *string* crua formatada, é possível utilizar ambos os prefixos simultaneamente: `rf'C:\{pasta}\{nome_arquivo}.txt'`.

10.3.2. Gerenciador de contexto

Antes de serguirmos para a explicação sobre as diferentes formas de se ler ou escrever um arquivo, vamos apresentar o gerenciador de contexto do Python, uma funcionalidade introduzida na versão 2.5 da linguagem, através da PEP 343 (ROSSUM, G. V., 2005).

O gerenciador de contexto é utilizado com o comando `with`, e serve para automatizar a abertura e fechamento de arquivos, garantindo que após sua execução, o arquivo será automaticamente fechado, mesmo que ocorra algum erro durante a execução dos comandos de leitura/escrita de dados.

Com ele, podemos reescrever o código da Codificação 10.1 como mostrado na Codificação 10.2. Aqui não precisamos nos preocupar com o fechamento do arquivo, isso será feito pelo próprio Python após o encerramento do bloco do comando `with`.

Codificação 10.2: Exemplo de abertura, processamento e fechamento de um arquivo com o uso do gerenciador de contexto do Python

```
with open('teste2.txt', 'w') as f:
    f.write('Olá novamente!\n')
```

Fonte: do autor, 2021

Faça o teste e veja que o arquivo "teste2.txt" foi criado com o texto do exemplo, e observe que não foi preciso fechar o arquivo. Podemos verificar isso através do atributo `closed` do objeto de arquivo criado no Python. Edite o código para corresponder à Codificação 10.3 e refaça o teste.

Codificação 10.3: Verificação do status de fechamento do arquivo com o atributo *closed*

```
with open('teste2.txt', 'w') as f:
    f.write('Olá novamente!\n')
    print(f'(dentro do bloco with) arquivo fechado? {f.closed}')

print(f'(fora do bloco with) arquivo fechado? {f.closed}')
```

Fonte: do autor, 2021

10.4. Arquivos de texto

É muito comum a utilização de arquivos de texto para diversos fins, seja para guardar informações em disco, como arquivos de log, ou para transmitir informações entre aplicações diferentes, que podem ou não estarem rodando na mesma máquina.

Então é natural que muitas linguagens de programação forneçam suporte a manipulação, isto é, leitura e escrita, de tais arquivos de maneira programática.

Ao abrirmos um arquivo de texto em Python, nos é retornado um objeto que é uma instância da classe `io.TextIOWrapper`, que faz parte do módulo `io`. Este módulo fornece diversas ferramentas para lidar com fluxos de entrada e saída de dados (*input* - *output*).

Esta classe herda de `io.TextIOBase`, que por sua vez herda de `io.IOBase`, portanto a lista completa de métodos disponíveis, e suas descrições, pode ser vista na documentação, conforme segue:

- `io.TextIOWrapper`: PSF (2021b);
- `io.TextIOBase`: PSF (2021c);
- `io.IOBase`: PSF (2021d).

É importante observar que o Python realiza 100% do processamento dos arquivos de texto nativamente, sem depender da forma como o sistema operacional opera sobre tais arquivos, de modo que este processamento é independente de plataforma.

10.4.1. Leitura de dados em arquivos texto

No momento que o arquivo é aberto para leitura, Python estabelece um marcador na posição zero do arquivo, indicando qual será o próximo caractere a ser lido, após a execução de uma instrução de leitura, este marcador é movido para uma nova posição, imediatamente após o último caractere lido pela instrução executada.

É possível manipular a posição desse marcador usando o método `seek` e é possível visualizar tal posição com o método `tell`, como mostrado no exemplo da codificação 10.4. No entanto, na prática, não é tão comum a utilização direta desses métodos.

Codificação 10.4: Utilização dos métodos *seek* e *tell*

```
with open('teste.txt', 'r') as f:
    print(f.tell())  # exibe a posição atual
    f.seek(5)        # move a posição atual para o índice 5
    print(f.tell())  # exibe a posição atual
```

Fonte: do autor, 2021

Para ler os dados de um arquivo podemos usar 3 métodos diferentes, como mostra as Codificação 10.5, 10.6 e 10.7:

- `f.read()`: lê todo o conteúdo do arquivo, a partir da posição atual do marcador até o final do arquivo (EOF), e retorna uma *string* com o conteúdo lido;

- `f.readline()`: lê o conteúdo do arquivo a partir da posição atual do marcador até encontrar uma quebra de linha, e move o marcador para o caractere seguinte ao da quebra de linha, isto é, para o primeiro caractere da linha seguinte, retornando também uma *string* com o conteúdo lido.
- `f.readlines()`: lê todo o conteúdo do arquivo, a partir da posição atual até o final do arquivo, mas agora retornando uma lista de *strings* na qual cada item corresponde a uma linha do arquivo.

Codificação 10.5: Leitura dos dados com o método *read*

```
with open('teste.txt', 'r') as f:
    texto = f.read()
# código que utiliza a variável texto após o bloco with
```

Fonte: do autor, 2021

Codificação 10.6: Leitura dos dados com o método *readlines*

```
with open('teste.txt', 'r') as f:
    linhas = f.readlines()
# código que utiliza a variável linhas após o bloco with
```

Fonte: do autor, 2021

Codificação 10.7: Leitura dos dados com o método *readline*

```
with open('teste.txt', 'r') as f:
    linha1 = f.readline()
    linha2 = f.readline()
    linha3 = f.readline()

# código que utiliza as variáveis após o bloco with
```

Fonte: do autor, 2021

Há também uma quarta forma de se ler um arquivo linha a linha, diretamente em um laço `for`, como mostra a Codificação 10.8. Para realizar o teste, crie um arquivo de texto, na mesma pasta que se encontra seu arquivo Python, e escreva algumas linhas nele. Se precisar de uma inspiração, copie o Zen Of Python, que pode ser obtido executando em uma Shell do Python: `>>> import this`, e cole nesse arquivo.

Codificação 10.8: Iterando sobre um arquivo diretamente

```
with open('teste.txt', 'r') as f:
    for linha in f:
        print(linha)
```

Fonte: do autor, 2021

Observe que nas Codificações 10.5, 10.6 e 10.7, o arquivo é fechado tão logo os dados são lidos, e o processamento destes dados pode ocorrer depois, sem deixar o arquivo "preso" ao processo do Python em execução. De maneira geral, essa abordagem é mais indicada, mas há situações em que pode ser necessário ou vantajoso a utilização

da abordagem feita na Codificação 10.8, em que não é possível fechar o arquivo e depois processar os dados, pois estamos operando diretamente sobre o arquivo e processando uma linha de cada vez, antes de ler a linha seguinte.

Uma situação em que este último exemplo pode ser vantajoso ocorre quando o arquivo é demasiado grande. Ao lermos uma linha de cada vez, não precisamos carregar o arquivo inteiro em memória antes de realizar o processamento. Em contrapartida, se diferentes partes da aplicação precisam acessar o mesmo arquivo, o melhor é fechar o arquivo o mais cedo possível para que ele não fique inacessível.

É importante observar que tentar ler um objeto Python de um arquivo que já foi lido irá resultar em uma string vazia, pois o marcador de posição já se encontrará no final do mesmo. Veja o exemplo da Codificação 10.9.

Codificação 10.9: Segunda leitura de um arquivo já lido

```
with open('teste.txt', 'r') as f:
    leitura1 = f.read()  # irá conter os dados do arquivo
    leitura2 = f.read()  # será uma string vazia
```

Fonte: do autor, 2021

Para reler o arquivo, podemos usar o método `seek` para voltar o marcador de posição para o começo do arquivo, ou fechá-lo e abri-lo novamente.

10.4.2. Escrita de dados em arquivos texto

A escrita de dados em um arquivo de texto pode ser feita de duas formas diferentes, como mostra os exemplos das Codificações 10.10 e 10.11:

- `f.write(texto)`: Escreve o conteúdo da variável `texto`, que deve ser uma *string*, no arquivo referenciado pela variável `f`.
- `f.writelines(s)`: Itera sobre a variável `s`, que deve ser uma sequência de *strings*, e escreve cada uma das strings no arquivo referenciado por `f`.

Primeiramente, vamos criar um arquivo novo no modo escrita, chamado “teste.py”, que irá sobrescrever o arquivo antigo, caso ele exista. Se preferir, pode usar outro nome para o arquivo.

Codificação 10.10: Escrita de arquivo com o método *write*

```
texto = 'escrevendo a primeira frase'
with open('teste.txt', 'w') as f:
    f.write(texto)
```

Fonte: do autor, 2021

Em seguida vamos abrir este mesmo arquivo, agora no modo `'a'`, para anexar o novo conteúdo ao final do arquivo, sem sobrescrevê-lo.

Codificação 10.11: Escrita de arquivo com o método *writelines*

```
s = ['linha 1', 'linha 2', 'linha 3', 'linha 4']
with open('teste.txt', 'a') as f:
    f.writelines(s)
```

Fonte: do autor, 2021

A utilização do método `write` é semelhante à função `print`, mas ao contrário do `print`, a escrita em um arquivo não irá adicionar automaticamente a quebra de linha. Da mesma forma, o método `writelines` tampouco irá adicionar quebras de linha entre os itens da sequência de *strings*, então, ao testar os exemplos acima, você irá observar que todo o texto saiu na mesma linha.

Para fazer com que os dados sejam escritos em linhas separadas, é necessário incluir na *string* que será escrita o caractere de quebra de linha. Vamos adicionar ao exemplo anterior a Codificação 10.12, que irá criar um novo arquivo de texto colocando as quebras de linha necessárias.

Codificação 10.12: Escrita de arquivo com o método *writelines*

```
texto2 = texto + '\n'
s2 = [f'{linha}\n' for linha in s]  # list comprehension[3]
with open('teste2.txt', 'w') as f:
    f.write(texto2)
    f.writelines(s2)
```

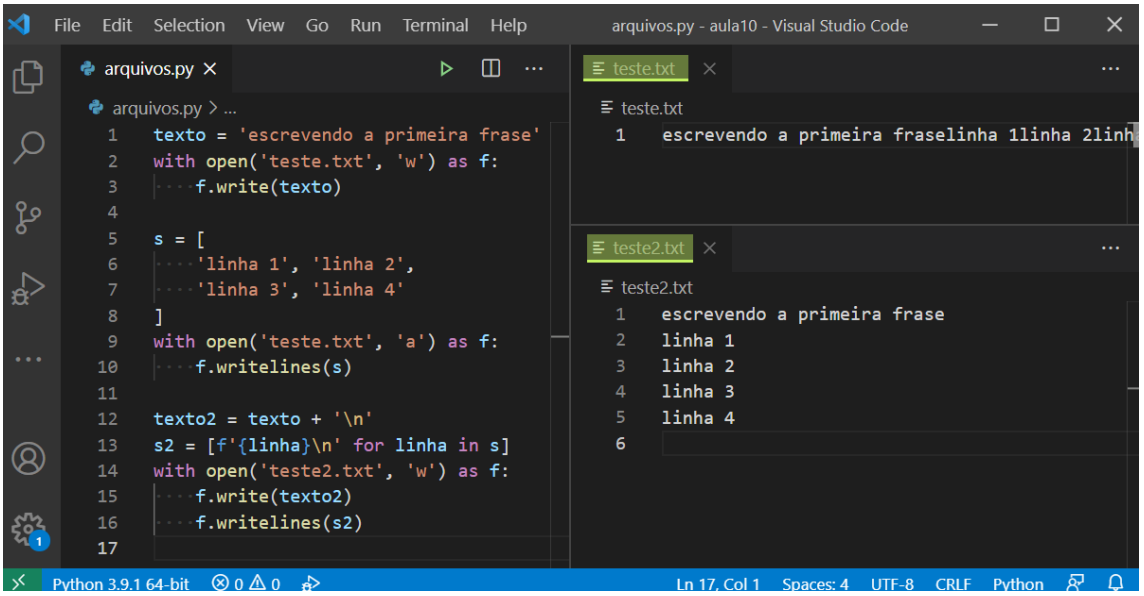
Fonte: do autor, 2021

Veja na Figura 10.1 o resultado da execução do código das Codificações 10.9 (incluindo a 10.12), 10.10 e 11.11, escrito no arquivo “arquivos.py”.

[3] Aqui utilizamos uma compreensão de listas para gerar a nova lista, isso é equivalente a fazer um laço `for` e adicionar cada item à nova lista, da seguinte maneira:

```
s2 = []
for linha in s:
    s2.append(f'{linha}\n')
```

Figura 10.1: Visualização dos arquivos gerados



Fonte: do autor, 2021

Há ainda uma outra forma de escrevermos um arquivo de texto usando a função print. Já vimos que podemos alterar o caractere de separação dos argumentos e de fim de linha ao fazermos uma exibição na tela com a função `print`, como na Codificação 10.13, ao passarmos valores (*strings*) para os parâmetros opcionais `sep` e `end`.

Codificação 10.13: Alteração do comportamento da função print com parâmetros opcionais nomeados.

```
>>> print(1, 2, 3, sep='...', end='!\n')
1...2...3!
>>>
```

Fonte: do autor, 2021

A função `print` possui mais um parâmetro opcional, que é o `file`. Por padrão esse parâmetro recebe a saída padrão (*stdout*) que, na maioria das plataformas, direciona para a tela do terminal ou Shell ativo, ou seja, a partir do qual o comando foi executado.

Se for passado um objeto de arquivo, como aquele gerado pela função `open`, o `print` irá então executar neste arquivo a "exibição" dos argumentos que recebeu. Veja na Codificação 10.14 como poderíamos reescrever o exemplo da Figura 10.1 usando a função `print`.

Codificação 10.14: Utilização da função *print* para escrita em arquivos de texto

```
texto = 'escrevendo a primeira frase'
s = ['linha 1', 'linha 2', 'linha 3', 'linha 4']
with open('print.txt', 'w') as f:
    print(texto, file=f)
    for linha in s:
        print(linha, file=f)
```

Fonte: do autor, 2021

10.4.3. Arquivos de texto especiais

Há diversos tipos de arquivos de texto, que podem ter outras extensões diferentes de "*.txt", para que sejam interpretados pelo sistema operacional (SO) por aplicações ou softwares específicos, mas que no fundo são arquivos de texto simples. Um exemplo desse tipo de arquivo são os próprios arquivos do Python, que terminam em "*.py" para que sejam identificados pelo SO como arquivos do Python, mas que são simplesmente arquivos de texto que contém código Python.

Dois destes tipos de arquivos dos quais é importante termos conhecimento são o CSV[4] e o JSON[5], que definem regras de sintaxe para estes arquivos de texto visando um objetivo final. No caso do CSV, este objetivo é a representação de tabelas e no caso do JSON é a troca de dados utilizando um determinado padrão de serialização de dados.

10.4.3.1. Arquivos CSV

Para trabalhar com arquivos CSV, o Python disponibiliza o módulo integrado `csv` (PSF, 2021e), que faz parte da instalação padrão do Python, mas precisa ser importado para ser utilizado. Entre outras, este módulo possui duas funções para leitura/escrita de dados a partir de sequências, e duas classes para leitura/escrita de dados a partir de dicionários. As funções e classes deste módulo possuem diversos parâmetros opcionais para configurar seu funcionamento, e o objetivo aqui será apenas fornecer um exemplo ilustrativo de uso.

Imagine que precisamos representar a Tabela 10.2 em Python, referente ao estoque de uma pequena loja de eletrodomésticos.

Tabela 10.2: Estoque de uma loja de eletrodomésticos. Fonte: Elaborado pelo autor

Descrição	Potência (Watts)	Tensão (Volts)	Quantidade em estoque	Preço (R$)
Liquidificador 5 velocidades	800	220	8	259,99

[4] CSV vem da sigla do inglês "*Comma-separated values*", que podemos traduzir para para "valores separados por vírgula".
[5] JSON vem da sigla em inglês "*JavaScript Object Notation*", que podemos traduzir para "notação de objetos do JavaScript".

Geladeira 350 litros	75	110	4	1372,99
Microondas 30 litros	1500	220	21	499,99

Fonte: do autor, 2021

A representação dessa tabela em memória poderia ser feita apenas com listas aninhadas, isto é, uma lista representativa da tabela, cujos itens sejam listas que representam cada produto, como indicado na Codificação 10.15.

Codificação 10.15: Representação da Tabela 10.2 em memória no Python

```
colunas = ['Descrição', 'Potência (Watts)', 'Tensão (Volts)',
           'Quantidade em estoque', 'Preço (R$)']
estoque = [
    ['Liquidificador 5 velocidades', 800, 220, 8, 259.99],
    ['Geladeira 350 litros', 75, 110, 4, 1372.99],
    ['Microondas 20 litros', 1500, 220, 21, 499.99],
]
```

Fonte: do autor, 2021

Essa é uma representação válida e útil para manipulação do estoque, por exemplo por funções que adicionam itens adquiridos para a loja ao estoque e retiram os itens vendidos aos clientes. No entanto, não é muito útil para mantermos um controle do estoque ao longo do tempo, pois ao final do dia, ao desligarmos o computador, os dados seriam perdidos (ou precisariam estar escritos no próprio código fonte, o que não é uma prática muito boa).

Uma alternativa é escrevê-los em um arquivo CSV, como na Codificação 10.16.

Codificação 10.16: Escrita de um arquivo CSV

```
import csv

with open('estoque.csv', 'w', newline='',
          encoding='utf-8') as arquivo_csv:
    escritor = csv.writer(arquivo_csv, delimiter=';')
    escritor.writerow(colunas)
    escritor.writerows(estoque)
```

Fonte: do autor, 2021

Primeiro precisamos abrir o arquivo no modo escrita e, para o caso de um arquivo CSV, recomenda-se sempre passar uma *string* vazia para o parâmetro `newline`. Isso é necessário pois o módulo `csv` faz seu próprio tratamento de novas linhas, e isso pode gerar problemas em plataformas que utilizam mais de um caractere para marcar o final de uma linha em arquivos de texto (como é o caso do Windows). Opcionalmente podemos também definir a codificação a ser usada no arquivo, para garantir que os caracteres serão escritos com a codificação desejada (se omitido, a codificação padrão do sistema operacional é usada).

Em seguida, no bloco do gerenciador de contexto, primeiro criamos um escritor, que será responsável por escrever os dados no arquivo e em seguida usamos os métodos `writerow` e `writerows`, que recebem respectivamente uma sequência de valores e uma sequência de sequências de valores, para escrever as linhas no arquivo.

Para ler o arquivo, o processo é semelhante, como mostra a Codificação 10.17.

Codificação 10.17: Leitura dos dados em um arquivo CSV

```
with open('estoque.csv', 'r', newline='',
          encoding='utf-8') as arquivo_csv:
    leitor = csv.reader(arquivo_csv, delimiter=';')
    novo_estoque = []
    for linha in leitor:
        novo_estoque.append(linha)

novas_colunas, *novo_estoque = novo_estoque         # 1
print(novas_colunas)
print('[', *novo_estoque, sep='\n  ', end='\n]')
```

Fonte: do autor, 2021

Abrimos o arquivo da mesma forma, criamos uma lista vazia para receber os dados que serão lidos pelo leitor criado, e por fim realizamos a leitura dos dados em um laço `for`. Em `#1`, estamos fazendo um desempacotamento da lista `novo_estoque`, colocando o primeiro item na lista `novas_colunas` e os demais sobrescrevendo a própria variável `novo_estoque`, apenas para facilitar a visualização no terminal. O mesmo resultado pode ser obtido com a Codificação 10.18.

Codificação 10.18: Forma alternativa de realizar o desempacotamento da lista

```
novas_colunas = novo_estoque[0]
novo_estoque = novo_estoque[1:]
```

Fonte: do autor, 2021

Como estamos apenas adicionando os elementos lidos a uma lista, sem fazer nenhum outro tipo de processamento, isso poderia ser realizado de maneira mais simples usando o construtor de lista para forçar a leitura dos dados e conversão para lista, como mostra a Codificação 10.19.

Codificação 10.19: Leitura dos dados usando o construtor de lista

```
with open('estoque.csv', 'r', newline='',
          encoding='utf-8') as arquivo_csv:
    leitor = csv.reader(arquivo_csv, delimiter=';')
    novo_estoque = list(leitor)
```

Fonte: do autor, 2021

O arquivo completo para escrever e ler o arquivo CSV pode ser visto na Figura 10.2, e o arquivo CSV gerado pode ser visto na Figura 10.3.

Figura 10.2: Código para escrever e ler um arquivo CSV

csv_.py > ...

```python
import csv

colunas = [
    'Descrição', 'Potência (Watts)', 'Tensão (Volts)',
    'Quantidade em estoque', 'Preço (R$)'
]
estoque = [
    ['Liquidificador 5 velocidades', 800, 220, 8, 259.99],
    ['Geladeira 350 litros', 75, 110, 4, 1372.99],
    ['Microondas 20 litros', 1500, 220, 21, 499.99],
]

with open('estoque.csv', 'w', newline='', encoding='utf-8') as arquivo_csv:
    escritor = csv.writer(arquivo_csv, delimiter=';')
    escritor.writerow(colunas)
    escritor.writerows(estoque)

with open('estoque.csv', 'r', newline='', encoding='utf-8') as arquivo_csv:
    leitor = csv.reader(arquivo_csv, delimiter=';')
    novo_estoque = []
    for linha in leitor:
        novo_estoque.append(linha)

novas_colunas, *novo_estoque = novo_estoque
print(novas_colunas)
print('[', *novo_estoque, sep='\n  ', end='\n]')

```

Fonte: do autor, 2021

Figura 10.3: Visualização do arquivo CSV no VSCode (acima) e no LibreOffice Calc (abaixo)

csv_.py | estoque.csv

estoque.csv

```
Descrição;Potência (Watts);Tensão (Volts);Quantidade em estoque;Preço (R$)
Liquidificador 5 velocidades;800;220;8;259.99
Geladeira 350 litros;75;110;4;1372.99
Microondas 20 litros;1500;220;21;499.99
```

	A	B	C	D	E
1	Descrição	Potência (Watts)	Tensão (Volts)	Quantidade em estoque	Preço (R$)
2	Liquidificador 5 velocidades	800	220	8	259.99
3	Geladeira 350 litros	75	110	4	1372.99
4	Microondas 20 litros	1500	220	21	499.99
5					

Fonte: do autor, 2021

10.4.3.2. Arquivos JSON

JSON é uma notação utilizada para representar objetos em JavaScript, e foi criada para funcionar como um formato leve de troca de dados entre diversas aplicações. Seu formato é completamente independente de qualquer linguagem de programação e sua estrutura é familiar a estruturas nativas de diversas linguagens, como por exemplo C, C++, C#, Java, JavaScript, Perl, Python e muitas outras (ECMA, 2017).

A sintaxe dos arquivos JSON faz com que sejam facilmente lidos tanto por computadores quanto por humanos e é um formato muito utilizado para a transmissão de dados na internet.

Um arquivo JSON é composto por duas estruturas: um mapeamento de pares chave-valor e uma sequência de itens ordenados, que, em Python, encontram uma tradução direta para o dicionário e a lista, respectivamente. Para trabalhar com arquivos JSON, o Python disponibiliza o módulo `json`, que, assim como vimos no módulo `csv`, faz parte da biblioteca padrão do Python.

10.4.3.2.1. Codificação Python para JSON

A Tabela 10.3 mostra a relação entre os tipos dados no Python e o tipo equivalente para o qual o dado será codificado ao ser serializado para o formato JSON.

Tabela 10.3: Tradução dos tipos de dados do Python para JSON (codificação)

Python	JSON
`dict`	`object`
`list, tuple`	`array`
`str`	`string`
`int, float e Enums`	`number`
`True`	`true`
`False`	`false`
`None`	`null`

Fonte: do autor, 2021

O módulo json define 2 funções para fazer a codificação de um objeto Python para JSON: `json.dump` e `json.dumps`. A primeira faz a serialização de um objeto Python para um arquivo JSON, ou arquivo binário com suporte ao método `.write()`, de acordo com tabela de conversão dada na Tabela 10.4. A segunda faz a mesma coisa, mas o resultado é devolvido como *string*[6].

Veja na Codificação 10.20 um arquivo que pega um dicionário em Python, representando os dados de um aluno, e faz duas conversões para JSON, em uma string e em um arquivo.

[6] Para ajudar a lembrar qual função escreve em um arquivo e qual retorna uma string, entenda o *s* no final da função como uma abreviação de *string*: dumps → *dump-string*.

Codificação 10.20: Conteúdo do arquivo “json_.py”

```
import json

aluno = {
    'nome': 'Paulo Ferreira',
    'ra': '001234567',
    'curso': 'ADS',
    'matriculado': True,
    'data_formatura': None,
    'disciplinas': [
        {
            'nome': 'Programação Orientada a Objetos',
            'notas_acs': [7, 8, 7.5],
            'nota_prova': 8,
        },
        {
            'nome': 'Desenvolvimento Web',
            'notas_acs': [10, 10, 7],
            'nota_prova': 9,
        },
        {
            'nome': 'Linguagem SQL',
            'notas_acs': [5, 9, 7.5],
            'nota_prova': 8.5,
        },
    ]
}

aluno_str = json.dumps(aluno, indent=2)
print(aluno_str)

with open('aluno.json', 'w') as f:
    json.dump(aluno, f, indent=2)
```

Fonte: do autor, 2021

Execute este e compare a saída na tela com o arquivo JSON gerado.

O uso do parâmetro `indent` é opcional, e foi feito para melhorar a visualização do arquivo por nós, humanos, se o arquivo será apenas lido por uma máquina, não há necessidade de utilizá-lo.

10.4.3.2.2. Decodificação JSON para Python

A Tabela 10.4 mostra a relação entre os tipos de dados no arquivo JSON e o tipo equivalente para o qual o dado será decodificado no Python quando desserializado.

Tabela 10.4: Tradução dos tipos de dados JSON para Python (decodificação)

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Fonte: do autor, 2021

O módulo json define também 2 funções para fazer a decodificação de um JSON para o Python: `json.load` e `json.loads`. A primeira faz a desserialização de um arquivo de texto, ou arquivo binário com suporte ao método `.read()`, contendo um documento JSON válido para um objeto Python, de acordo com tabela de conversão dada na Tabela 10.3. A segunda faz a mesma coisa mas a partir de um objeto de *string*[7].

Adicione ao arquivo da Codificação 10.20 o conteúdo da codificação 10.21.

Codificação 10.21: Continuação do arquivo "json_.py"[8]

```
aluno_carregado_da_string = json.loads(aluno_str)

with open('aluno.json', 'r') as f:
    aluno_carregado_do_arquivo = json.load(f)

print('Comparação 1:', aluno_carregado_da_string == aluno)
print('Comparação 2:', aluno_carregado_do_arquivo == aluno)
```

Fonte: do autor, 2021

Ao executarmos novamente o arquivo "json_.py", observamos que os dicionários carregados a partir tanto da *string* como do arquivo são de fato iguais ao objeto inicial. Mas é importante ressaltar que isso nem sempre é verdade, pois, ao converter um

[7] Podemos aplicar o mesmo pensamento aqui, leia o *s* no final da função como *load-string*.
[8] O nome do arquivo foi pós fixado com um sublinhado para evitar conflito com o módulo json do Python, pois, como vimos, todo arquivo Python é automaticamente interpretado como um módulo local.

dicionário para JSON, todas as chaves do dicionário são convertidas para string e listas e tuplas viram *array*, na ida, e na volta *array* é convertido em lista.

Então caso o dicionário possua chaves que não sejam strings e seja convertido para JSON e depois de volta para dicionário, não será igual ao original, isto é, `loads(dumps(x)) != x`, já que as chaves no dicionário convertido de volta para o Python irão permanecer como *strings*.

10.5. Arquivos binários

A manipulação de arquivos binários (PSF, 2021g) é útil para todos os tipos de arquivo que não sejam de texto, como por exemplo, imagens e vídeos, e pode ser útil também para arquivos de texto, como por exemplo criar uma cópia criptografada do arquivo original ou gerar um arquivo comprimido. Vamos ilustrar aqui como podemos escrever um arquivo binário na memória usando Python.

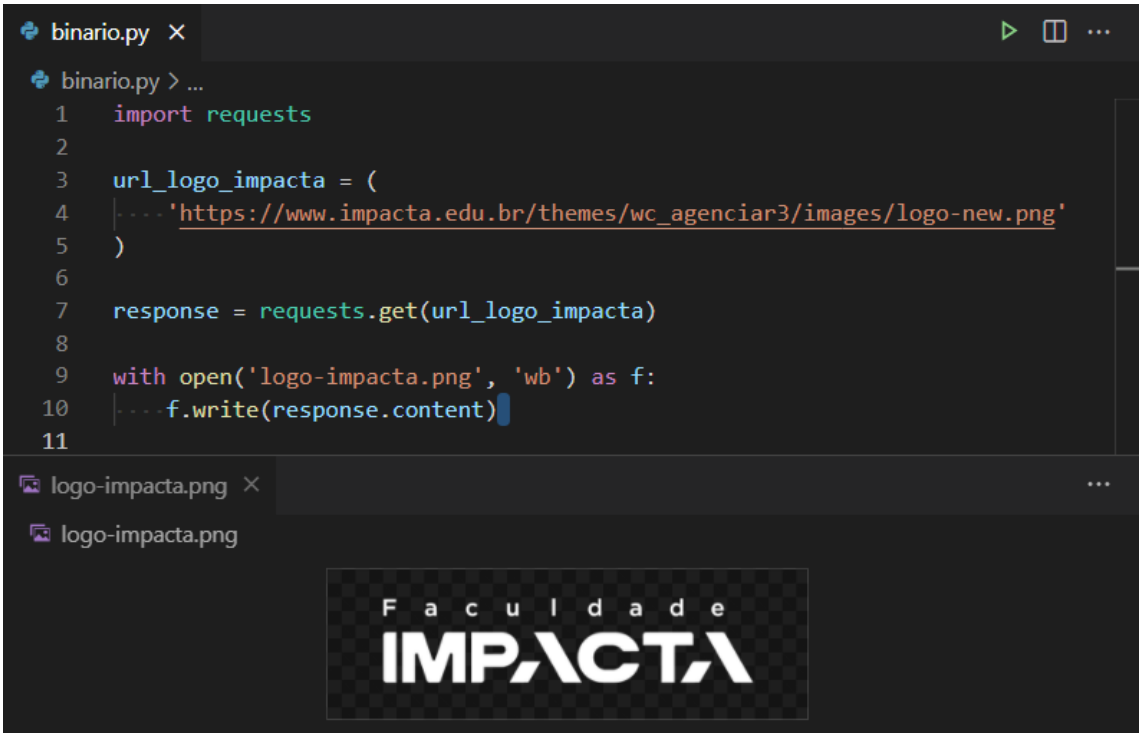
Codificação 10.17: Escrita de um arquivo binário com Python

```
import requests
url_logo_impacta = (
 'https://www.impacta.edu.br/themes/wc_agenciar3/images/logo-new.png'
)
response = requests.get(url_logo_impacta)
with open('logo-impacta.png', 'wb') as f:
    f.write(response.content)
```

Fonte: do autor, 2021

Para que o código acima funcione, é necessário instalar o módulo `requests`, o que pode ser feito com o comando: `> pip install requests`. Este módulo é usado para fazer requisições HTTP e usamos o método get para fazer uma requisição GET ao endereço público do logo da Impacta. Em seguida acessamos o conteúdo da resposta e escrevemos-o em um arquivo binário, aberto com o modo `'wb'`, que representa a escrita em binário. O resultado pode ser visto na Figura 10.4.

Figura 10.4: Logo da Impacta salvo em um arquivo png



Fonte: do autor, 2021

10.6. Tópicos relacionados

O Python possui várias outras ferramentas embutidas na biblioteca padrão para trabalhar com arquivos e realizar as mais diversas tarefas relacionadas ao acesso de arquivos e diretórios, a persistência de dados em arquivos e à compressão de dados e arquivamento, como por exemplo:

- Caminhos do sistema de arquivos orientados a objetos, com o módulo pathlib;
- Manipulações comuns de nomes de caminhos do sistema de arquivos, com o módulo os.path;
- Serialização de objetos Python, com o módulo pickle;
- Interface nativa para o banco de dados SQLite, com o módulo sqlite3;
- Suporte a compressão de arquivos (zip e tar) e módulos para os algoritmos de compressão LZMA e bzip2.

A lista completa e atualizada pode ser vista na documentação (PSF, 2021h; PSF, 2021i; PSF, 2021j).

Bibliografia

ECMA INTERNATIONAL. **ECMA 404**: the JSON data interchange syntax. 2017. Dispnível em: <ECMA-404 - Ecma International (ecma-international.org)>. Acesso em: 24 abr. 2021.

PSF. **Funções embutidas**: open. 2021a. Disponível em: <https://docs.python.org/3/library/functions.html#open>. Acesso em: 23 abr. 2021.

PSF. **Ferramentas principais para trabalhar com fluxos**: io.TextIOWrapper. 2021b. Disponível em: <https://docs.python.org/pt-br/3/library/io.html#io.TextIOWrapper>. Acesso em: 23 abr. 2021.

PSF. **Ferramentas principais para trabalhar com fluxos**: io.TextIOBase. 2021c. Disponível em: <https://docs.python.org/pt-br/3/library/io.html#io.TextIOBase>. Acesso em: 23 abr. 2021.

PSF. **Ferramentas principais para trabalhar com fluxos**: io.IOBase. 2021d. Disponível em: <https://docs.python.org/pt-br/3/library/io.html#io.IOBase>. Acesso em: 23 abr. 2021.

PSF. **Leitura e escrita de arquivos CSV**. 2021e. Disponível em: <https://docs.python.org/pt-br/3/library/csv.html>. Acesso em: 23 abr. 2021.

PSF. **Codificador e decodificador JSON**. 2021f. Disponível em: <https://docs.python.org/pt-br/3/library/json.html>. Acesso em: 23 abr. 2021.

PSF. **Ferramentas para leitura e escrita de arquivos binários**. 2021g. Disponível em: <https://docs.python.org/3/library/io.html#binary-i-o>. Acesso em: 23 abr. 2021.

PSF. **Acesso a arquivos e diretórios**. 2021h. Disponível em: <https://docs.python.org/pt-br/3/library/filesys.html>. Acesso em: 24 abr. 2021.

PSF. **Persistência de dados**. 2021i. Disponível em: <https://docs.python.org/pt-br/3/library/persistence.html>. Acesso em: 24 abr. 2021.

PSF. **Compressão de dados e arquivamento**. 2021j. Disponível em: <https://docs.python.org/pt-br/3/library/archiving.html>. Acesso em: 24 abr. 2021.

ROSSUM, G. V.; COGHLAN, N. **PEP 343**: the “with” Statement. 2005. Disponível em: <https://www.python.org/dev/peps/pep-0343/#abstract>. Acesso em: 23 abr. 2021.