Revisão de Python: Manipulação de Arquivos

> Arquivos são usados para armazenar dados em um dispositivo de armazenamento permanente.

- Os arquivos podem ser de texto ou binários.
  - **Texto:** podem ser editados com um editor de texto.
  - **Binários:** precisam ser manipulados com um programa específico.

Criando arquivos de texto

> Arquivos de texto são usados para armazenar dados legíveis por humanos. Ex.: arquivos .txt , .c, .py, .html, .xml, .json, etc

- **Criação de Arquivos:** A função `open()` é usada para criar um objeto para acessar um arquivo.

```
f = open("arquivo.txt", "w")
f.close()
```

- **Modos de Abertura:** A função `open()` aceita um segundo argumento que especifica o modo de abertura do arquivo.
  - `r` : Abre um arquivo para leitura. O arquivo deve existir (padrão).
  - `w` : Abre um arquivo para escrita. Se o arquivo não existir, ele será criado. Se o arquivo existir, ele será sobrescrito.
  - `a` : Abre um arquivo para anexar. Se o arquivo não existir, ele será criado. Se o arquivo existir, os dados serão anexados ao final do arquivo.
  - `x` : Cria um novo arquivo. Se o arquivo já existir, a operação falhará.

> Os modos de abertura podem ser combinados com os seguintes caracteres:

- `t` : Abre um arquivo em modo texto (padrão).
- `b` : Abre um arquivo em modo binário.

- **Close:** A função `close()` é usada para fechar um arquivo.

> É importante fechar um arquivo após a leitura ou escrita para liberar recursos do sistema operacional e garantir que os dados sejam gravados corretamente.

- **Verificação de Fechamento:** A função `closed` é usada para verificar se um arquivo está fechado.

```
f = open("arquivo.txt", "w")
print(f.closed)
f.close()
print(f.closed)
```

- **Escrita em Arquivos:** A função `write()` é usada para escrever em um arquivo.

```
f = open("arquivo.txt", "w")
f.write("Olá, Mundo!")
f.close()
```

> Para escrever em um arquivo, é necessário abrir o arquivo em um modo que permita a escrita (por exemplo, `w` ou `a`).

- **Adição de Conteúdo:** A função `write()` é usada para adicionar conteúdo a um arquivo.

```
f = open("arquivo.txt", "a")
f.write("Olá, Mundo!")
f.close()
```

> O conteúdo é adicionado ao final do arquivo.

- **flush():** A função `flush()` é usada para antecipar a gravação de dados em um arquivo.

```
f = open("arquivo.txt", "w")
# (...) suposto processo demorado sujeito a falhas
for i in range(1000):
    f.write(str(i))
    f.flush() # Grava os dados imediatamente
f.close()
```

> Deve ser usado em situações em que é importante garantir que os dados sejam gravados imediatamente, mesmo que o arquivo não seja fechado.

Leitura de Arquivos de Texto

- A função `read()` é usada para ler um arquivo.

```
f = open("arquivo.txt")
conteudo = f.read()
f.close()
print(conteudo)
```

> read() lê todo o conteúdo do arquivo e o armazena em uma string. No entanto, se o arquivo for muito grande, isso pode consumir muita memória. Para evitar isso, é possível ler o arquivo linha por linha ou em pedaços menores.

- Parâmetro `size`: O parâmetro `size` é usado para especificar o número máximo de bytes a serem lidos.

```
f = open("arquivo.txt")
while True:
    conteudo = f.read(10) # Lê 10 bytes por vez
    if not conteudo:
        break
    print(conteudo)
f.close()
print(conteudo)
```

> O método `read()` retorna uma *string* vazia quando o final do arquivo é atingido.

- **Leitura por Linhas:** A função `readline()` é usada para ler uma linha de cada vez.

```
f = open("arquivo.txt")
linha = f.readline()
while linha:
    print(linha)
    linha = f.readline()
f.close()
```

- **Leitura de Linhas:** A função `readlines()` é usada para ler todas as linhas de uma vez e armazená-las em uma lista.

```
f = open("arquivo.txt")
linhas =
for linha in linhas:
    print(linha)
f.close()
```

- **Uso do `with`**: O bloco `with` é usado para garantir que o arquivo seja fechado corretamente.

```
with open("arquivo.txt") as f:
    conteudo = f.read()
    print(conteudo)
```

> O bloco `with` garante que o arquivo seja fechado automaticamente após a execução do bloco, mesmo se ocorrer uma exceção.

> Também é possível usar `finnaly` para garantir que o arquivo seja fechado, mas o bloco `with` é mais elegante.

```
f = open("arquivo.txt")
try:
    conteudo = f.read()
    print(conteudo)
finally:
    f.close()
```

Acesso Randomizado

> Embora seja raro, é possível acessar um arquivo de forma randomizada, saltando para uma posição específica.

- **Movimentação do Cursor:** O método `seek()` é usado para mover o cursor para uma posição específica.

```
with open("arquivo.txt", "r") as f:
    f.seek(5)
    print(f.read())
```

- **Posição do Cursor:** O método `tell()` é usado para obter a posição atual do cursor.

```
with open("arquivo.txt", "r") as f:
    l = f.readline()
    print(f.tell())
```

Arquivos de Texto com Dados Estruturados

> Arquivos de texto são comumente usados para armazenar dados estruturados.
> Esses dados são usados para testar algoritmos e programas.

- Exemplos:
  - O site The Matrix Market é um repositório de dados de matrizes esparsas.
  - O site The DIMACS Graph Format é um repositório de dados de problemas de otimização.
  - O site TSPLIB é um repositório de dados de problemas de otimização.

> A estrutura dos arquivos de texto varia de acordo com a fonte dos dados.

- Para ler esses dados, é necessário entender a estrutura do arquivo que está, geralmente, descrita no site onde os dados estão disponíveis.
- Para algumas fontes muito populares, como o TSPLIB, existem bibliotecas que podem ser usadas para ler os dados.

Exemplo formato matrix market:

```
%%MatrixMarket matrix coordinate real general
%-------------------------------------------------------------------------
% UF Sparse Matrix Collection, Tim Davis
% http://www.cise.ufl.edu/research/sparse/matrices/
% name: Gleich/dolphins
% [Matrix Market, IJV] graph: Gleich/dolphins
%-------------------------------------------------------------------------
62 62 159
1 1 1.000000e+00
1 2 1.000000e+00
1 3 1.000000e+00
...
```

Exemplo formato TSPLIB:

```
NAME: a280
TYPE: TSP
COMMENT: drilling problem (Ludwig)
DIMENSION: 280
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 288 149
2 288 129
3 270 133
...
```

Arquivos de Texto Separados por Vírgula (CSV)

> Arquivos CSV (comma-separated values) são usados para armazenar dados tabulares.

- Cada linha do arquivo é uma linha da tabela.
- Cada valor é separado por vírgula.
- O primeiro linha pode ser um cabeçalho.

Exemplo de arquivo CSV:

```
Nome,Função,Salário
Edu,Engenheiro,10000
Bob,Programador,8000
Rui,Designer,6000
Ana,"Art,Visual",7500
```

> Obs.: Se uma string contiver uma vírgula, ela será colocada entre aspas duplas.

> Embora seja possível ler arquivos CSV manualmente, é mais prático usar uma biblioteca para fazer isso.

A biblioteca `csv` é usada para ler e escrever arquivos CSV.

- A função `reader()` é usada para ler um arquivo CSV.

```
import csv
with open("arquivo.csv", "r") as arquivo:
    leitor = csv.reader(arquivo)
    for linha in leitor:
        print(linha)
```

- A função `writer()` é usada para escrever um arquivo CSV.

```
import csv
with open("arquivo.csv", "w") as arquivo:
    escritor = csv.writer(arquivo)
    escritor.writerow(["Nome", "Idade"])
    escritor.writerow(["Albert", 30])
```

Arquivos com Campos de Largura Fixa

Uma forma comum de armazenar dados é usar campos de largura fixa.

- *Fixed Width Text File*
- Cada linha do arquivo é uma linha da tabela.
- Cada campo tem uma quantidade fixa de caracteres.
- O primeiro linha pode ser um cabeçalho.

Exemplo de arquivo com campos de largura fixa:

```
NAME                 STATE      TELEPHONE
John Smith           WA         418-Y11-4111
Mary Hartford        CA         319-Z19-4341
Evan Nolan           IL         219-532-c301
```

Ver mais

- É possível usar slicing para ler arquivos com campos de largura fixa.
- A função `strip()` é usada para remover espaços em branco.

```
with open("arquivo.txt", "r") as f:
    for linha in f:
        nome = linha[0:20].strip()
        estado = linha[20:30].strip()
        telefone = linha[30:40].strip()
        print(nome, estado, telefone)
```

> A fonte de dados deve especificar o número de caracteres de cada campo.

- Para escrever arquivos com campos de largura fixa, é possível usar a função `format()`.

```
with open("arquivo.txt", "w") as arquivo:
    arquivo.write("{:<10}{:02}{:06}\n".format("RUI", 30,3000))
```

- Um exemplo de uso de campos de largura fixa é o histórico da Bolsa de Valores.

```
00COTAHIST.2023BOVESPA 20231228
012023010202NEOE3       010NEOENERGIA  ON      NM   R$  0000000001540000000000154000000000014980000000001508000000000150300000000015030000000001506019160000000000032500000000000490215100000000000000009999123100000010000000000000BRNEOEACNOR3117
012023010202NEXP3       010NEXPE       ON      NM   R$  0000000000031000000000003100000000000290000000000029000000000000300000000000030000000000003100255000000000001977000000000000058980000000000000000099991231000000100000000000000BRNEXPACNOR0100
```

> para o ano de 2023, o arquivo de histórico diário da Bovespa tem mais de 500MB.

Histórico B3

Descrição do arquivo

Arquivos JSON

Arquivos JSON (JavaScript Object Notation) são usados para armazenar dados estruturados.

- Surgiu como uma solução para a comunicação entre servidores web e navegadores.
- É fácil de ler e escrever.
- É suportado nativamente pelo Python.

> A biblioteca `json` é usada para ler e escrever arquivos JSON.

- A função `dump()` é usada para escrever um arquivo JSON.

```
import json
dados = {}
dados["Edu"]= {"idade": 30, "salario": 10000}
dados["Bob"]= {"idade": 25, "salario": 8000}
dados["Rui"]= {"idade": 30, "salario": 6000}
with open("arquivo.json", "w") as arquivo:
    json.dump(dados, arquivo)
```

- A função `load()` é usada para ler um arquivo JSON.

```
import json
with open("arquivo.json", "r") as arquivo:
    dados = json.load(arquivo)
    print(dados)
```

Exemplo de arquivo JSON:

```
{
    "Edu": {"idade": 30, "salario": 10000},
    "Bob": {"idade": 25, "salario": 8000},
    "Rui": {"idade": 30, "salario": 6000}
}
```

> Um outro formato de arquivo estruturado é o XML. Ao contrário do JSON, este formato é complexo e não é nativo do Python. Não abordaremos este formato aqui.

Ver mais

Arquivos Binários

> Arquivos binários são usados para armazenar dados não legíveis por humanos.

- Vantagens:
  - Ocupam menos espaço.
  - Leia e escreva mais rapidamente.
  - Ocultam informações sensíveis.
- Desvantagens:
  - Não legíveis por humanos.
  - Difíceis de manipular sem um programa específico.
- Exemplos:
  - jpg, png, mp3, mp4, exe, dll, zip, pdf, etc.

- Escrevendo uma lista de números inteiros em um arquivo binário, um número por vez.

```
numeros = [1, 2, 3, 4, 5]
with open("arquivo.bin", "wb") as f:
    for numero in numeros:
        f.write(numero.to_bytes(4, "little"))
```

- A função `to_bytes()` é usada para converter um número inteiro em bytes. O primeiro argumento é o número de bytes e o segundo argumento é a ordem dos bytes. O número de bytes deve ser suficiente para armazenar o número inteiro.
- A ordem dos bytes pode ser "little" (menos significativo primeiro) ou "big" (mais significativo primeiro).

- Lendo uma lista de números inteiros de um arquivo binário.

```
numeros = []
with open("arquivo.bin", "rb") as f:
    while True:
        numero = f.read(4)
        if not numero:
            break
        numeros.append(int.from_bytes(numero, "little"))
print(numeros)
```

- A função `from_bytes()` é usada para converter bytes em um número inteiro.

- Escrevendo dados tabulares em um arquivo binário.

```
dados = {}
dados["Edu"]= {"idade": 30, "salario": 10000}
dados["Bob"]= {"idade": 25, "salario": 8000}
dados["Rui"]= {"idade": 30, "salario": 6000}
with open("arquivo.bin", "wb") as f:
    for nome, info in dados.items():
        #string com comprimento fixo
        f.write(f'{nome[:20]:20}'.encode("utf-8"))
        f.write(info["idade"].to_bytes(4, "little"))
        f.write(info["salario"].to_bytes(4, "little"))
```

- A função `encode()` é usada para converter uma string em bytes.

- Lendo dados tabulares de um arquivo binário.

```
dados = {}
with open("arquivo.bin", "rb") as f:
    while True:
        nome = f.read(20).decode("utf-8").strip()
        if not nome:
            break
        idade = int.from_bytes(f.read(4), "little")
        salario = int.from_bytes(f.read(4), "little")
        dados[nome] = {"idade": idade, "salario": salario}
print(dados)
```

> Obs.: A fonte de dados deve especificar o número de bytes de cada campo.

A biblioteca `pickle` é usada para serializar e desserializar objetos Python, ou seja, converter objetos Python em bytes e vice-versa.

- A função `dump()` é usada para escrever um arquivo binário.

```
import pickle
dados = {}
dados["Edu"]= {"idade": 30, "salario": 10000}
dados["Bob"]= {"idade": 25, "salario": 8000}
dados["Rui"]= {"idade": 30, "salario": 6000}
with open("arquivo.bin", "wb") as f:
    pickle.dump(dados, f)
```

- A função `load()` é usada para ler um arquivo binário.

```
import pickle
with open("arquivo.bin", "rb") as f:
    dados = pickle.load(f)
    print(dados)
```

Manipulando Arquivos no Sistema de Arquivos

- O módulo `os` é usado para manipular arquivos no sistema de arquivos.
- O módulo `os.path` é usado para manipular caminhos de arquivos.
- O módulo `shutil` é usado para manipular arquivos e diretórios.
- O módulo `glob` é usado para encontrar arquivos que correspondem a um padrão.
- O módulo `pathlib` é usado para manipular caminhos de arquivos.

- **Verificando a Existência de um Arquivo:** A função `exists()` é usada para verificar se um arquivo existe.

```
import os
print(os.path.exists("arquivo.txt"))
```

- **Verificando se um Arquivo é um Diretório:** A função `isdir()` é usada para verificar se um arquivo é um diretório.

```
import os
print(os.path.isdir("arquivo.txt"))
```

- **Verificando se um Arquivo é um Arquivo:** A função `isfile()` é usada para verificar se um arquivo é um arquivo.

```
import os
print(os.path.isfile("arquivo.txt"))
```

- **Renomeando um Arquivo:** A função `rename()` é usada para renomear um arquivo.

```
import os
os.rename("arquivo.txt", "novo_arquivo.txt")
```

- **Movendo um Arquivo:** A função `rename()` é usada para mover um arquivo.

```
import os
os.rename("arquivo.txt", "diretorio/novo_arquivo.txt")
```

- **Copiando um Arquivo:** A função `copy()` é usada para copiar um arquivo.

```
import shutil
shutil.copy("arquivo.txt", "novo_arquivo.txt")
```

- **Deletando um Arquivo:** A função `remove()` é usada para remover um arquivo.

```
import os
os.remove("arquivo.txt")
```

- **Deletando um Diretório:** A função `rmdir()` é usada para remover um diretório.

```
import os
os.rmdir("diretorio")
```

- **Deletando um Diretório com Tudo Dentro:** A função `rmtree()` é usada para remover um diretório com subdiretórios e arquivos.

```
import shutil
shutil.rmtree("diretorio")
```

- **Criando um Diretório:** A função `mkdir()` é usada para criar um diretório.

```
import os
os.mkdir("diretorio")
```

- **Criando um Diretório com Subdiretórios:** A função `makedirs()` é usada para criar um diretório com subdiretórios.

```
import os
os.makedirs("diretorio/subdiretorio")
```

- **Listando Arquivos em um Diretório:** A função `listdir()` é usada para listar arquivos em um diretório.

```
import os
print(os.listdir("diretorio"))
```

- **Encontrando Arquivos com um Padrão:** A função `glob()` é usada para encontrar arquivos que correspondem a um padrão.

```
import glob
print(glob.glob("*.txt"))
```

> O padrão `*` corresponde a qualquer número de caracteres.
> Encontrando arquivos com a extensão `.txt` no diretório atual.

- **Encontrando Arquivos com um Padrão Recursivamente:** A função `glob()` é usada para encontrar arquivos que correspondem a um padrão recursivamente.

```
import glob
print(glob.glob("**/*.txt", recursive=True))
```

> O padrão `**` corresponde a qualquer número de diretórios.
> Encontrando arquivos com a extensão `.txt` em todos os subdiretórios.

- **Manipulando Caminhos de Arquivos:** O módulo `pathlib` é usado para manipular caminhos de arquivos.

```
from pathlib import Path
caminho = Path("diretorio/arquivo.txt")
print(caminho.name) # arquivo.txt
print(caminho.suffix) # .txt
print(caminho.parent) # diretorio
print(caminho.stem)# arquivo
```

Outros recursos

Não abordaremos aqui, mas é possível:

- Manipular arquivos compactados.
  - bibliotecas: `zipfile`, `tarfile`, `gzip`, `bz2`, `lzma`, `zstd`.
- Criar arquivos criptografados.
  - bibliotecas: `cryptography`, `pycryptodome`.
- Criar arquivos temporários.
  - biblioteca: `tempfile`.
- Criar arquivos de log.
  - biblioteca: `logging`.

Conclusão

- Arquivos são usados para armazenar dados em um dispositivo de armazenamento permanente.
- Arquivos podem ser de texto ou binários.
- Python tem inumeras bibliotecas para manipular arquivos, cada uma com seu propósito.

Exercícios

1. Sem usar biblioteca específica para tsplib, escreva uma função que leia um arquivo da TSP LIB e imprima o número de cidades, as coordenadas de cada cidade e retorne a matriz de distâncias euclidianas entre as cidades.
2. Usando o resultado do exercício anterior, escreva uma função que salve a matriz de distâncias em um arquivo CSV.
3. Escreva uma função que leia um arquivo CSV da questão anterior e cria um arquivo **binário** com os dados. Comente sobre a diferença de tamanho entre os arquivos.
4. Sem usar biblioteca específica para bovespa, escreva uma função que leia um arquivo de histórico da Bovespa e imprima o nome do ação mais negociada no dia do seu aniversário (ou data mais próxima).