Interagindo com o sistema de arquivos

Vamos ao terminal do Linux e explorar algumas das operações de filesystem e aprender como efetuamos as mesmas operações via Python.

Criar uma pasta

No Linux

```
$ mkdir pasta1
```

No Python

```
>>> import os
>>> os.mkdir("pasta2")
```

Acessar uma pasta

No linux

```
$ cd pasta1
```

No Python

```
>>> import os
>>> os.chdir("pasta2")
```

Exibir a pasta atual

No Linux

```
$ pwd
/path/pasta1
```

No Python

```
>>> import os
>>> os.path.abspath(os.curdir)
/path/pasta2
```

Criar um arquivo em branco

No linux

```
$ touch arquivo.txt
```

No Python

```
>>> open("arquivo.txt", "w")
```

Listar arquivos

No linux:

```
$ ls
arquivo.txt
```

No Python

```
>>> import os
>>> os.listdir(".")
['arquivo.txt']
```

Escrever em um arquivo

No Linux

```
$ echo "Hello" >> arquivo.txt
```

> **nota** `>` escreve no modo `w` substituindo todo o conteúdo do arquivo e `>>` escreve no modo `a` fazendo append no final do arquivo existente.

No Python

```
>>> arquivo =  open("arquivo.txt", "a")
>>> arquivo.write("Hello\n")
>>> arquivo.close()
```

> **nota** `"w"` escreve no modo `w` substituindo todo o conteúdo do arquivo e `"a"` escreve no modo `a` fazendo append no final do arquivo existente.

Ler o conteúdo de um arquivo

No Linux

```
$ cat arquivo.txt
Hello
```

No Python

```
>>> print(open("arquivo.txt", "r").read())
'Hello\n'
```

Dicas úteis ao trabalhar com arquivos no Python

Context manager

Nós ainda não falamos sobre este tema, mas para trabalhar com arquivos ele é essencial, no exemplo de escrita em arquivos nós tivemos que chamar a função `.close()` explicitamente.

```
>>> arquivo =  open("arquivo.txt", "a")
>>> arquivo.write("Hello\n")
>>> arquivo.close()
```

Como é muito importante manter os descritores de arquivo devidamente encerrados em Python sempre ao abrir um arquivo iremos dar preferência para o uso de um context manager, que é um bloco especial de código que automaticamente executa operações como o `.close` em arquivos.

A maneira preferida será sempre:

```
with open("arquivo.txt", "a") as arquivo:
    # aqui temos o arquivo aberto para escrever

    arquivo.write("Hello")
```

```
    # aqui o context manager garante o fechamento do arquivo
    # sem a necessidade de chamarmos explicitamente o .close()
```

Criar diretórios não existem em um caminho

```
>>> os.mkdir("um/outro/maisoutro/)
FileNotFoundError: [Errno 2] No such file or directory: 'um/outro/maisoutro/'
```

Para o código acima funcionar teriamos que criar primeiro `um` e depois `outro` e assim por diante, portanto quando temos multiplas pastas para criar vamos preferir usar a função `makedirs`

```
>>> os.makedirs("um/outro/maisoutro", exist_ok=True)
>>> os.listdir("um")
['outro']
>>> os.listdir("um/outro")
['maisoutro']
```

Ler multiplas linhas de um arquivo

Um arquivo de texto pode ter milhares de linhas e ao efetuar a leitura com `.read` isso pode causar um estouro de memória:

```
>>> open("arquivo_grande.txt", "r").read()
Error: Not Enough Memory to load 20GB
```

Neste caso devemos usar o procolo de iteração que já aprendemos na nossa aula de listas, os file descriptors são iteráveis, e melhor que isso eles são geradores de dados.

Isso quer dizer que dentro de um loop, a leitura do arquivo linha a linha será mais eficiente:

```
for linha in open("arquivo_grande.txt", "r"):
    print(linha)
```

Vai imprimir tranquilamente pois mesmo que o arquivo tenha `20GB` será carregado para a memória apenas uma linha de cada vez.

Escrever multiplas linhas em um arquivo

```
texto = [
    "Este é um texto",
    "cada item desta lista",
    "é uma linha no arquivo",
]

with open("arquivo.txt", "a") as arquivo:
    arquivo.writelines("\n".join(texto))
```

No linux pode ler com:

```
$ cat arquivo.txt
Este é um texto
cada item desta lista
é uma linha no arquivo
```

Lendo multiplas linhas de um arquivo

> Lembrando que isso só deve ser feito em arquivos de tamanho pequeno.

```
>>> open("arquivo.txt").readlines()
['Este é um texto\n', 'cada item desta lista\n', 'é uma linha no arquivo']
```

Trabalhando com caminhos

```
# Juntando caminhos
>>> import os
>>> os.path.join("pasta2/pasta3/arquivo.txt")
pasta2/pasta3/arquivo.txt

# Obtendo caminho absoluto
>>> os.path.abspath(os.path.join("pasta2/pasta3/arquivo.txt"))
/home/user/pasta2/pasta3/arquivo.txt

# Obtendo o caminho absoluto para o diretorio atual
>>> os.path.abspath(os.curdir)
/tmp/pasta2/foo/
```

A Pathlib

A pathlib foi adicionada no Python 3 e provê as mesmas funcionalidades do `os` e `open` com algumas melhorias de sintaxe.

```
>>> from pathlib import Path
# Criar pasta
>>> Path("pasta3").mkdir(parents=True, exist_ok=True)
# Criar arquivo na pasta
>>> Path("pasta3/arquivo.txt").touch()
# Escrever no arquivo
>>> Path("pasta3/arquivo.txt").write_text("Bruno")
# Ler o conteúdo do arquivo
>>> Path("pasta3/arquivo.txt").read_text()
'Bruno'
```

Uma caracteristica interessante do objeto `Path` é que ele permite ser combinado com outros objetos do mesmo tipo ou `str` usando `/`

```
>>> caminho = Path("pasta1") / "pasta2" / Path("pasta3")
>>> print(caminho)
pasta1/pasta2/pasta3
```

Conclusão

A escolha entre utilizar `os` ou `pathlib` se dá pelo gosto de quem está programando as funcionalidades são as mesmas e na maioria dos casos as funções executadas serão as mesmas, cabe a quem estiver programando verificar qual das interfaces está lidando na hora de interagir com os objetos file descriptors.

Exercicios

Salvar o histórico de cálculos da calculadora infixcalc.py em um arquivo informado no parametro `--logfile`

Alterar o script `interpolacao.py` para ler emails de um arquivo `emails.txt` ao enviar os emails.

Criar um bloco de anotações `python3 note.py write` que pergunta `titulo`, `tag`, `texto` salva em uma nova linha de um arquivo chamado `notas.txt` separados por `tab` (tsv) ao executar com `python3 note.py read` exibe todas as notas linha a linha permitindo filtrar com `--tag=geral`

Guia de referencia:

Modes

- `'r'` - Read (default).
- `'w'` - Write (truncate).
- `'x'` - Write or fail if the file already exists.
- `'a'` - Append.
- `'w+'` - Read and write (truncate).
- `'r+'` - Read and write from the start.
- `'a+'` - Read and write from the end.
- `'t'` - Text mode (default).
- `'b'` - Binary mode.

Exceptions

- `'FileNotFoundError'` can be raised when reading with `'r'` or `'r+'`.
- `'FileExistsError'` can be raised when writing with `'x'`.
- `'IsADirectoryError'` and `'PermissionError'` can be raised by any.
- `'OSError'` is the parent class of all listed exceptions.

File Object

```
<file>.seek(0)                      # Moves to the start of the file.
<file>.seek(offset)                 # Moves 'offset' chars/bytes from the start.
<file>.seek(0, 2)                   # Moves to the end of the file.
<bin_file>.seek(±offset, <anchor>)  # Anchor: 0 start, 1 current position, 2 end.
```

```
<str/bytes> = <file>.read(size=-1)  # Reads 'size' chars/bytes or until EOF.
<str/bytes> = <file>.readline()     # Returns a line or empty string/bytes on EOF.
<list>      = <file>.readlines()    # Returns a list of remaining lines.
<str/bytes> = next(<file>)          # Returns a line using buffer. Do not mix.
```

```
<file>.write(<str/bytes>)           # Writes a string or bytes object.
<file>.writelines(<collection>)     # Writes a coll. of strings or bytes objects.
<file>.flush()                      # Flushes write buffer.
```