Lendo e Escrevendo Arquivos

Asimov Academy

ASIMOV

Conteúdo

01. O que vamos aprender neste curso

Bem vindo ao curso Lendo e Escrevendo Arquivos da Asimov Academy!

A Memória de uma Linguagem de Programação

Você deve estar vindo de um curso básico de Python, nele aprendeu a salvar valores em variáveis. Isso é o primeiro aspecto da memória de uma linguagem de programação. Seria como a memória de curto prazo, pois ela existe apenas no período em que o código está sendo rodado.

Mas e agora se quero armazenar um valor por um período mais longo, por dias e anos. Para que sempre que eu inicie o meu código ele lembre do ponto que parou? Para isso, teremos que ativar a memória de longo prazo do nosso programa, que é exatamente o salvamento de informações no nosso disco rígido.

Memória RAM x Disco Rígido

A diferença de disco rígido e memória ram: para você ter um entendimento melhor, vou fazer uma brevíssima explicação de dois conceitos que são fundamentais na computação. A memória ram e o disco de armazenamento. A memória ram é nossa memória de curto prazo no computador, ela é muito rápida e pode ser facilmente acessada, mas ela tem um tamanho pequeno. Já o disco de armazenamento possui um tamanho maior de armazenamento, mas é mais lento para ser acessado.

Os nossos scripts salvam os valores na memória ram, portanto quanto mais memória ram tiver seu pc, maior a capacidade de rodar scripts mais pesados, com mais informações. Só lembrando, essa memória é pequena e o propósito é utilizá-la no curto prazo. Portanto, quando desligamos um script a memória ram que ele consumia é limpa e essas variáveis desaparecem

Para aumentarmos as capacidades de memória do nosso Script, podemos utilizar o disco rígido. Nele armazenamos informações perenes. Uma das forams mais simples de fazer isso é através do salvamento de arquivos. Arquivos simples, como esses que utilizamos no nosso dia a dia, como csv, xlsx, txt, etc

Esse curso portanto se propõem a ensinar como expandir as capacidades de memória do nosso script, como mantê-la por longos prazos e como organizar e manipular essa memória dentro do nosso computador.

Vamos aprender a ler e escrever diversos tipos de dados, como txt, xlsxl, csv, pickle e também como movimentar esses arquivos por pastas, criar pastas para melhor organização e terminaremos com um projeto grande para fixar o aprendizado.

02. Arquivos e Caminho de Arquivos

- Todo arquivo tem duas propriedades: nome e caminho

Nome do Arquivo

Todo arquivo possui um nome específico pelo qual ele pode ser reconhecido. Esse nome sempre termina com .algo. Esse ponto algo representa a extensão do arquivo e avisa ao usuário e ao sistema operacional qual é o tipo do arquivo. A partir desse tipo, o sistema sabe com qual programa deve abrir cada arquivo. Por exemplo, ele sabe que para abrir um arquivo docx ele precisa utilizar um software como o Word. E é o que normalmente ocorre quando damos duplo clique em um arquivo. A extensão é importante para a utilização do arquivo por Python, pois para cada tipo de arquivo há uma forma mais adequada de abrir e salvar quando utilizamos um script

Caminho do Arquivo

O caminho (path) especifica o local dentro do computador em que o arquivo se encontra. O computador possui uma estrutura de pastas e arquivos. Na hierarquia, todo arquivo está dentro de uma pasta. Todas as pastas contém outras pastas ou arquivos. Já os arquivos são como a folha de uma árvore, são o ponto final. As pastas são como os galhos que podem conter outros galhos ou folhas.

A raiz, ou seja, a pasta que contém tudo que está no seu computador é chamada de C: no Windows. Mas para cada sistema operacional o nome é diferente.

O caminho é formado pela sequência de pastas desde a pasta raiz que seguidas sequencialmente chegam ao arquivo de interesse.

Na próxima aula vamos tratar de como construir caminho corretamente em Python para que consigamos localizar com facilidade nossos arquivos.

03. Construindo caminhos com pathlib

É importante lembrar que inglês a palavra path significa justamente caminho. Logo, a biblioteca que utilizaremos para criar caminhos de diretórias é justamente a pathlib (biblioteca de caminhos).

É uma biblioteca padrão do Python e não precisa de instalação

Vamos dar um exemplo:

```
from pathlib import Path

print(Path('primeira_pasta/segunda_pasta'))
# primeira_pasta\\segunda_pasta

print(type(Path('primeira_pasta/segunda_pasta')))
# <class 'pathlib.WindowsPath'>
```

Barra no mac e linux e dupla barra invertida no Windows

ATENÇÃO: Em Windows, caminhos são criados utilizando barra invertida (\) para separação das pastas. Já macOS e Linux utilizam a barra simples (/). Por isso é importante a utilização de Path(), pois ele tornará seu script compatível com todos os sistemas operacionais.

Criando manualmente caminhos para meus arquivos

Então vamos para um exemplo, digamos que temos três arquivos diferentes no seguinte caminho: **C:\Users\Al**. Uma forma de criar o caminho de um arquivo é a seguinte:

```
from pathlib import Path

for nome in ['arquivo1.txt', 'arquivo2.txt', 'arquivo3.txt']:
    print(Path('primeira_pasta/segunda_pasta', nome))
#'primeira_pasta\\segunda_pasta\\arquivo1.txt
#'primeira_pasta\\segunda_pasta\\arquivo2.txt
#'primeira_pasta\\segunda_pasta\\arquivo3.txt
```

Outra forma possível é utilizando barra para unir caminhos:

```
from pathlib import Path

for nome in ['arquivo1.txt', 'arquivo2.txt', 'arquivo3.txt']:
    print(Path('primeira_pasta/segunda_pasta') / nome)
```

```
#'primeira_pasta\\segunda_pasta\\arquivo1.txt
#'primeira_pasta\\segunda_pasta\\arquivo2.txt
#'primeira_pasta\\segunda_pasta\\arquivo3.txt
```

O Python executa esse comando da esquerda para direita. Portanto, o primeiro ou o segundo valores devem ser um objeto de Path.

```
homePath = Path('C:/Users/Al')
pasta = Path('spam')
subPasta = 'pasta'
print(homePath / pasta / subPasta)
#'C:\\Users\\Al\\spam'
```

Se nenhum dos dois primeiros for path, resultaremos em um erro:

```
homePath = 'C:/Users/Al'
pasta = 'spam'
subPasta = Path('pasta')
print(homePath / pasta / subPasta)
#'C:\\Users\\Al\\spam'
```

O diretório Home

Os sistemas operacionais são construídos de forma que cada usuário possui uma pasta para os arquivos do usuário. No nosso dia a dia, utilizamos quase sempre pastas e arquivos que estão dentro do diretório home. Por exemplo, os meus documentos, downloads, imagens, todas essas pastas ficam dentro do home. É importante saber localizar esse home com Python, pois em geral ele fará parte do caminho quando procurarmos o caminho absoluto de um arquivo. Para isso, podemos utilizar:

```
print(Path.home())
```

Seus scripts provavelmente terão permissão para escrever dentro desse diretório. Portanto, é um ótimo local para colocar os arquivos que seu programa de python utilizará.

04. Caminhos Absolutos e Caminhos Relativos

Existe duas formas para especificar o caminho de um arquivo: **caminho absoluto** e **caminho relativo**

Caminho Absoluto

É o caminho que sempre inicia pelo diretório raiz (C:\ no caso do Windows). É o caminho completo. Ele mostra todos os passos que você teria que fazer pelo seu *filesystem* para localizar um arquivo.

Caminho Relativo

Antes de explicarmos o funcionamento dos caminhos relativos precisamos explicar um conceito importante de Python, chamado **diretório de trabalho** (do inglês, *working directory*)

Todo o programa rodando em seu computador roda a partir de um diretório, o tal do diretório de trabalho. O pathlib possui um método para verificar qual é o diretório de trabalho atual do nosso programa:

```
from pathlib import Path

print(Path.cwd())
```

O cwd significa *current working directory*, ou seja, diretório atual de trabalho. Podemos trocar o nosso script de diretório de trabalho, utilizando o método chdir do módulo os.

```
from pathlib import Path

print(Path.cwd())

os.chdir(Path.home())

print(Path.cwd())
```

O comando chdir significa change directory, ou seja, muda diretório. Ele troca para o diretório informado, como Path.home() retorna o caminho para nossa pasta home, a partir desse comando conseguimos mudar nosso diretório de trabalho para a pasta home.

Fica assumido que todos os arquivos e pastas que tentamos localizar sem utilizar o diretório raiz estão abaixo do **diretório de trabalho**.

Vamos mostrar o funcionamento de caminhos absolutos e relativos através de exemplos: - Exemplo comparando o nome de arquivos pelo caminho absoluto e pelo caminho relativo - Mostrar como encontrar arquivos que estão antes hierarquicamente em relação ao diretório de trabalho - O que acontece quando rodo um arquivo de python em pastas diferentes e como o diretório de trabalho muda - Como fica o diretório de trabalho quando abrimos arquivos de diferentes formas no VSCode - Exemplos de listdir rodando para diferentes níveis de diretório de trabalho

05. Manipulando Caminhos de Arquivos

Primeiro vamos verificar se um caminho passado é relativo ou absoluto

```
Path.cwd()
#WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
Path.cwd().is_absolute()
#True
Path('primeira_pasta').is_absolute()
#False
```

Para transformar um path relativo em absoluto é simples

```
Path('primeira_pasta')
#WindowsPath('primeira_pasta')
Path.cwd() / Path('primeira_pasta')
#WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37/primeira_pasta')
```

Caso esse caminho seja relativo em relação a outro arquivo (como no exemplo, relativo ao home), podemos utilizar o seguinte:

```
Path('primeira_pasta')
#WindowsPath('primeira_pasta')
Path.home() / Path('primeira_pasta')
#WindowsPath('C:/Users/Al/primeira_pasta')
```

Muitas vezes colocamos arquivos na mesma pasta onde criamos nosso script. Para garantir que o script sempre encontre o arquivo, podemos utilizar a variável especial __file__:

```
from pathlib import Path

print(__file__)
print('caminho absoluto: ', Path( __file__ ).absolute())
print('caminho absoluto para a pasta: ', Path( __file__ ).parent.absolute())
```

Pegando Partes de um Caminho

Podemos dividir um caminho absoluto em algumas partes básicas: - A âncora,que é justamente o diretório raiz - O *drive* que no *Windows* é a letra que representa o disco que está sendo usado pela máquina - O pai (ou *parent* em inglês) que representa a pasta que contém o arquivo - O nome do arquivo que é formado pelo: - O nome base ou *steam* - E o sufixo, ou extensão

```
p = Path('C:/Users/Al/spam.txt')

print(p.anchor)
#'C:\\'
```

```
print(p.parent)
#WindowsPath('C:/Users/Al')

print(p.name)
#'spam.txt'

print(p.stem)
#'spam'

print(p.suffix)
#'.txt'

print(p.drive)
#'C:'
```

Podemos encontrar outros diretórios que constituem o caminho através do *parents*

```
p = Path('C:/Users/Al/spam.txt')

print(p.parents[0])
#WindowsPath('C:/Users/Al')

print(p.parents[1])
#WindowsPath('C:/Users')

print(p.parents[2])
#WindowsPath('C:')
```

06. Retornando Conteúdos de uma Pasta

Podemos facilmente listar os arquivos e diretórios de uma pasta com os seguintes comandos:

```
print(os.listdir(Path.home()))
```

Podemos utilizar também apenas o módulo pathlib para obter o mesmo resultado:

```
print(Path.home().glob('*'))
```

O método glob utilza a notação também chamada glob. Quando utilizamos * estamos pedindo para listar tudo dentro da pasta. Podemos selecionar extensões específicas, podemos fazer da seguinte forma:

```
print(Path.home().glob('*.txt'))
```

Dessa forma ele só listará os arquivos de extensão .txt

Validando caminhos

O fato de escrevermos um caminho não significa que ele exista no nosso computador. Para verificar se ele de fato existe, podemos utilizar o seguinte:

```
pasta_nao_existe = Path('C:/This/Folder/Does/Not/Exist')
pasta_existe = Path.home()

print('Não existe:', pasta_nao_existe.exists())
#Não existe: False
print('Existe:', pasta_existe.exists())
#Não existe: True
```

Podemos verificar também se o arquivo é uma pasta ou um arquivo:

```
pasta_existe = Path.home()

print('É uma pasta:', pasta_existe.is_dir())
#É uma pasta: True
print('É um arquivo:', pasta_existe.is_file())
#É uma arquivo: False
```

07. EXERCÍCIO - Encontrando arquivo

Exercício

Desenvolva um script que procura por toda a pasta home um arquivo com o nome que o usuário solicitou

```
def encontra_arquivo(nome_do_arquivo, nome_exato=True):
    for arquivo in Path.home().glob('**/*'):
        if arquivo.is_file():
            if ((nome_exato and nome_do_arquivo == arquivo.name)
                or (not nome_exato and nome_do_arquivo in arquivo.name)) :
                print(arquivo.absolute())
```

```
def encontra_arquivo(nome_do_arquivo, nome_exato=True, caminho_de_inicio=Path.home()):
    for arquivo in caminho_de_inicio.glob('*'):
        if arquivo.is_file():
            if ((nome_exato and nome_do_arquivo == arquivo.name)
                or (not nome_exato and nome_do_arquivo in arquivo.name)) :
                print(arquivo.absolute())
        else:
            encontra_arquivo(nome_do_arquivo, nome_exato, caminho_de_inicio=arquivo)
```

08. DESAFIO - Tamanho de uma pasta

Vamos criar um script para verificar o tamanho de todas as pastas que estão localizadas no diretório home:

```
def retorna_tamanho_dos_diretorios(caminho):
    for diretorio in Path(caminho).glob('*'):
        if diretorio.is_dir() and not diretorio.name.startswith('.'):
            tamanho_diretorio = 0
            for arquivo in diretorio.glob('**/*'):
                if arquivo.is_file():
                    tamanho_diretorio += os.path.getsize(arquivo)
            print(diretorio.name, round(tamanho_diretorio / 1024 / 1024))

retorna_tamanho_dos_diretorios(Path.home())
```

```
def retorna_tamanho_dos_diretorios(caminho, profundidade=1, tamanho_linha=0):
    for diretorio in Path(caminho).glob('*'):
        if diretorio.is_dir() and not diretorio.name.startswith('.'):
            tamanho_diretorio = 0
            for arquivo in diretorio.glob('**/*'):
                if arquivo.is_file():
                    tamanho_diretorio += os.path.getsize(arquivo)
            print('--' * tamanho_linha, diretorio.name, round(tamanho_diretorio / 1024 / 1024))
            if profundidade > 1:
                retorna_tamanho_dos_diretorios(diretorio, profundidade-1, tamanho_linha+1)
retorna_tamanho_dos_diretorios(Path.home(), profundidade=2)
```

09. Copiando, Movendo e Deletando Arquivos

Já sabemos como criar caminhos que permitem encontrar os nossos arquivos. Agora vamos começar a organizá-los!

Existem dois módulos de Python que são utilizadas para esse tipo de operação: **os** e **shutil**. Ambos conseguem executar as mesmas operações, mas nesse curso vamos focar na utilização de shutil.

Shutil é um módulo específico para operações de arquivos e é construido sobre o módulo os. Isso quer dizer que, por debaixo dos panos, o próprio shutil utiliza operações do módulo os.

Então, por que utilizar shutil?

Por que ela já uma biblioteca de alto nível, preparada para usuários que não querem se preocupar com algumas especificidades. Para fazer uma transferência de arquivos, shutil pode utilizar uma dúzia de métodos diferentes do os e outros módulos para garantir que essa transferência ocorra com sucesso. Essas garantias, para quem utiliza o módulo os diretamente, teriam que ser feita pelo próprio usuário na hora de desenvolver seu script. Para 99% dos casos, sugiro o uso de shutil. Apenas para programadores que quiserem um acesso mais fino a sua aplicação, sugiro o módulo os para esse tipo de operação. ___ ## Copiando Arquivos

A primeira forma é utilizando o método copyfile. Esse método não preserva as permissões nem metadados do arquivo.

```
from pathlib import Path
import shutil

pasta_atual = Path( __file__ ).parent
caminho_arquivo = pasta_atual / 'texto.txt'
caminho_arquivo_destino = pasta_atual / 'destino1' / 'texto.txt'

shutil.copyfile(caminho_arquivo, caminho_arquivo_destino)
```

A outra forma é com copy. Diferente de copyobject, o destino é a pasta no qual o arquivo vai ser copiado e não o caminho completo com nome. O método copy copia também as permissões do arquivo

```
from pathlib import Path
import shutil

pasta_atual = Path( __file__ ).parent.absolute()
caminho_arquivo = pasta_atual / 'texto.txt'

caminho_pasta_destino = pasta_atual / 'destino2'

shutil.copy(caminho_arquivo, caminho_pasta_destino)
```

Pro último podemos usar o copy2. A forma de uso é igual ao copy, mas ele preserva tanto permissões quanto metadados de

```
from pathlib import Path
import shutil

pasta_atual = Path( __file__ ).parent.absolute()
caminho_arquivo = pasta_atual / 'texto.txt'

caminho_pasta_destino = pasta_atual / 'destino2'

shutil.copy2(caminho_arquivo, caminho_pasta_destino)
```

Movendo Arquivos

Podemos utilizar o método move. Ele pode receber tanto o arquivo de destino quanto a pasta de destino

```
from pathlib import Path
import shutil

pasta_atual = Path( __file__ ).parent.absolute()
caminho_arquivo = pasta_atual / 'texto.txt'
caminho_arquivo_destino = pasta_atual / 'destino1' / 'texto.txt'

shutil.move(caminho_arquivo, caminho_arquivo_destino)
```

```
from pathlib import Path
import shutil

pasta_atual = Path( __file__ ).parent.absolute()
caminho_arquivo = pasta_atual / 'texto.txt'

caminho_pasta_destino = pasta_atual / 'destino2'

shutil.move(caminho_arquivo, caminho_pasta_destino)
```

Deletando Arquivos

```
from pathlib import Path
import shutil
import os

pasta_atual = Path( __file__ ).parent.absolute()
caminho_arquivo = pasta_atual / 'texto.txt'
caminho_arquivo_destino = pasta_atual / 'destino1' / 'texto.txt'

shutil.copyfile(caminho_arquivo, caminho_arquivo_destino)

if caminho_arquivo_destino.exists():
    os.remove(caminho_arquivo_destino)
```

10. Copiando, Movendo e Deletando Pastas

Algumas funções de criação de pastas não existem no shutil. Por isso, podemos novamente utilizar o pathlib.

Criando Pastas

Para criação de um único diretório, podemos utilizar o comando mkdir:

```
import os
from pathlib import Path

pasta_atual = Path( __file__ ).parent
caminho_pasta_destino = pasta_atual / 'destino4'
caminho_pasta_destino.mkdir()
```

Use `exist_ok=True` para caso a pasta já exista e não queira que gere um exceção!

Se queremos criar além da pasta destino, todas as páginas intermediárias até ela, podemos utilizar o mkdir com `parents=True`:

```
import os
from pathlib import Path

pasta_atual = Path( __file__ ).parent.absolute()
caminho_pasta_destino = pasta_atual / 'destino5' / 'destino51'
caminho_pasta_destino.mkdir(parents=True)
```

Copiando Pastas

Para mover uma pasta com todo conteúdo que está dentro dela, podemos usar o copytree do shutil:

```
import os
import shutil
from pathlib import Path

pasta_atual = Path( __file__ ).parent.absolute()

shutil.copytree(pasta_atual / 'destino4', pasta_atual / 'destino1' / 'destino4',
↪  dirs_exist_ok=True)
```

Deletando Pastas

Para remover uma pasta vazia podemos utilizar o rmdir

```
import os
from pathlib import Path

pasta_atual = Path( __file__ ).parent.absolute()
caminho_pasta_destino = pasta_atual / 'destino4' / 'destino41'
caminho_pasta_destino.rmdir()
```

Para remover uma pasta com todo conteúdo que está dentro dela, podemos usar o rmtree do shutil:

```
import os
import shutil
from pathlib import Path

pasta_atual = Path( __file__ ).parent.absolute()
caminho_pasta_destino = pasta_atual / 'destino4' / 'destino41'
caminho_pasta_destino.mkdir(parents=True)

shutil.rmtree(caminho_pasta_destino.parent)
```

11. Compactando e Descompactando Pastas

Podemos também utilizar o shutil para compactar e descompactar arquivos. Isso é uma funcionalidade interessante para caso queiramos criar um script para fazer backup de nossos dados por exemplo. Poderíamos semanalmente rodar, compactar os arquivos que nos são de interesse e enviá-lo para um disco externo por exemplo, mantendo nesse disco um cópia segura de nossos dados.

```
import shutil
from pathlib import Path

pasta_atual = Path( __file__ ).parent
nome_do_arquivo = pasta_atual / 'compactado'
pasta_que_sera_compactada = pasta_atual


shutil.make_archive(nome_do_arquivo, 'zip', pasta_que_sera_compactada)
```

Para descompactar, podemos utilizar o seguinte método

```
import shutil
from pathlib import Path

pasta_atual = Path( __file__ ).parent.absolute()
nome_do_arquivo = pasta_atual / 'compactado.zip'
pasta_que_sera_descompactada = pasta_atual / 'descompactado'


shutil.unpack_archive(nome_do_arquivo, pasta_que_sera_descompactada, 'zip')
```

12. DESAFIO - Organizando Arquivos por extensão

Organizando arquivos por formato. Crie pastas diferentes para cada formato de arquivo (por exemplo, arquivo .pdf deve ir para pasta pdf), depois de organizado, faça um zip contendo todos os arquivos, sendo o nome do arquivo a data de hoje.

```
from pathlib import Path
import shutil
import datetime


pasta_atual = Path(__file__).parent
pasta_a_organizar = pasta_atual / 'arquivos_desafio'
pasta_organizada = pasta_atual / 'organizada'
pasta_backups = pasta_atual / 'backups'


if pasta_organizada.exists():
    shutil.rmtree(pasta_organizada)
pasta_organizada.mkdir()

if not pasta_backups.exists():
    pasta_backups.mkdir()

for arquivo in pasta_a_organizar.glob('**/*'):
    if arquivo.is_file():
        pasta_com_extensao = pasta_organizada / arquivo.suffix.replace('.', '')
        if not pasta_com_extensao.exists():
            pasta_com_extensao.mkdir()
        shutil.copy(arquivo, pasta_com_extensao)

nome_backup = datetime.datetime.now().strftime('%Y_%m_%d')
shutil.make_archive(pasta_backups / nome_backup, 'zip', pasta_organizada)
```

13. Lendo e Escrevendo Arquivos de Texto

Agora já sabemos movimentar arquivos entre pastas, então estamos prontos para finalmente começar a criar arquivos que possamos utilizar nos nossos scripts para armazenar dados.

O tipo de arquivo mais simples são os arquivos de texto e por isso começaremos com eles.

Antes, precisamos falar de algumas particularidades que podem gerar alguns problemas para nós.

Quebra de Linhas

Primeiro, as quebras de linha são interpretadas diferentemente entre Windows e Linux/mac. Por exemplo, esse arquivo criado em Windows:

```
Pug\r\n
Jack Russell Terrier\r\n
English Springer Spaniel\r\n
German Shepherd\r\n
Staffordshire Bull Terrier\r\n
Cavalier King Charles Spaniel\r\n
Golden Retriever\r\n
West Highland White Terrier\r\n
Boxer\r\n
Border Terrier\r\n
```

Será interpretado de forma diferente pelo Linux:

```
Pug\r
\n
Jack Russell Terrier\r
\n
English Springer Spaniel\r
\n
German Shepherd\r
\n
Staffordshire Bull Terrier\r
\n
Cavalier King Charles Spaniel\r
\n
```

```
Golden Retriever\r
\n
West Highland White Terrier\r
\n
Boxer\r
\n
Border Terrier\r
\n
```

Isso por que a quebra de linha é indicada de forma diferente entre os dois sistemas

Codificação (encoding) de Caracteres

Outro problema comum que você pode enfrentar é a codificação dos dados do byte. Uma codificação é uma tradução de dados de byte para caracteres legíveis por humanos. Isso geralmente é feito atribuindo um valor numérico para representar um caractere. As duas codificações mais comuns são os formatos ASCII e UNICODE. O ASCII pode armazenar apenas 128 caracteres, enquanto o Unicode pode conter até 1.114.112 caracteres.

ASCII é, na verdade, um subconjunto de Unicode (UTF-8), o que significa que ASCII e Unicode compartilham os mesmos valores numéricos para caracteres. É importante observar que a análise de um arquivo com a codificação de caracteres incorreta pode levar a falhas ou deturpação do caractere. Por exemplo, se um arquivo foi criado usando a codificação UTF-8 e você tentar analisá-lo usando a codificação ASCII, se houver um caractere fora desses 128 valores, um erro será gerado.

Mais sobre: https://www.ime.usp.br/~pf/algoritmos/apend/unicode.html

Abrindo um Arquivo

O Python fornece uma função built-in para a leitura de arquivos, a função open:

```
lista_compras = open('lista_de_compras.txt')
```

É importante lembrar que, sempre que abrimos um arquivo é como se tivéssemos aberto uma planilha no computador. Temos que explicitamente pedir para o Python fechá-lo. Caso contrário, ele ficará aberto e consumindo memória ram do seu programa. É um erro comum esquecer de fechar o arquivo, o que acarreta em diversos problemas no script.

Então vamos aprender a fechar um arquivo:

```
lista_compras.close()
```

Outra forma mais utilizada (e mais recomendada) é utilizar o **with**.

```
with open('lista_de_compras.txt') as lista_compras:
    # processamentos do arquivo aqui
```

Nesse caso, ocorre que o objeto criado pela função open é armazenado na variável apostila(da mesma forma de quando fazemos *apostila = open('apostila.txt')*). A diferença é que quando saímos da identação do with, automaticamente o arquivo é fechado e o objeto deixa de existir. Isso faz com que a variável só exista então dentro do escopo do with, e não precisamos portanto utilizar o close, garantindo que nosso arquivo esteja aberto apenas quando estamos utilizando ele.

Modos de abertura de um arquivo Existem diversos modos nos quais os arquivos podem ser abertos, e fica a cargo do usuário decidir qual se adéqua mais às necessidades do seu código. Por exemplo, utilizando o parâmetro 'r', abriremos um arquivo no modo de leitura, não sendo permitido realizar modificações nele.

```
with open('lista_de_compras.txt', 'r') as lista_compras:
    # processamentos do arquivo aqui
```

Segue uma tabela dos principais modos:

Caracteres	Significado
`'r'`	Modo apenas leitura (padrão)
`'r+'`	Modo de leitura e escrita
`'w'`	Modo de escrita, reescreve arquivo caso já exista com o mesmo nome
`'a'`	Acrescenta a um arquivo sem reescrever conteúdo preexistente
`'rb'`	Faz a leitura do arquivo em binário

Para esclarecer melhor, podemos adicionar essa tabela que

Permissão	r	r+	w	w+	a	a+
leitura	+	+		+		+
escrita		+	+	+	+	+
cria			+	+	+	+
sobrescreve			+	+		
posição no início	+	+	+	+		

Permissão	r	r+	w	w+	a	a+
posição no final					+	+

- leitura: permite a leitura do arquivo
- escrita: permite a escrita no arquivo
- cria: cria um arquivo novo caso ele não exista
- sobrescreve: sobrescreve o arquivo caso ele exista
- posição no início: após aberto, a posição inicial da leitura do arquivo é colocada no início
- posição no final: após aberto, a posição inicial da leitura do arquivo é colocada no final

Lendo um arquivo

Existe três métodos principais de se ler um arquivo:

Método	Uso
`.read()`	Ele lê todo o arquivo de uma só vez
`.readline()`	Ele uma linha completa do arquivo
`.readlines()`	Lê as linhas remanescentes e retorna como uma lista de linhas

Utilizando os métodos:

```
from pathlib import Path
pasta_atual = Path( __file__ ).parent.absolute()

with open(pasta_atual / 'lista_de_compras.txt', 'r') as lista_compras:
    print(lista_compras.read())

with open(pasta_atual / 'lista_de_compras.txt', 'r') as lista_compras:
    linha = lista_compras.readline()
    while linha != '':
        print(linha, end='')
        linha = lista_compras.readline()

with open(pasta_atual / 'lista_de_compras.txt', 'r') as lista_compras:
    print(lista_compras.readlines())
```

Escrevendo em um arquivo

Existe dois métodos principais de se escrever em um arquivo:

Método	Uso
`.write(texto)`	Escreve a variável texto (que deve ser uma string) ao arquivo
`.writelines(linhas)`	Escreve uma sequência de linhas ao arquivo. Recebe uma lista de valores strings como argumento

No exemplo a seguir, atualizamos a lista de compras retirando itens que já foram comprados. Primeiro utilizamos o método write, que recebe uma string completa como argumento.

```
itens_ja_comprados = ['ovos', 'leite']

with open(pasta_atual / 'lista_de_compras.txt', 'r') as lista_compras:
    itens_lista_compras = lista_compras.readlines()

with open('lista_de_compras_atualizada.txt', 'w') as lista_atualizada:
    for item in itens_lista_compras:
        if not item.replace('\n', '') in itens_ja_comprados:
            lista_atualizada.write(item)
```

Já no segundo exemplo, utilizamos o método writelines, que recebe uma lista de strings como argumento.

```
itens_ja_comprados = ['ovos', 'leite']

with open(pasta_atual / 'lista_de_compras.txt', 'r') as lista_compras:
    itens_lista_compras = [item for item in lista_compras.readlines() \
                           if not item.replace('\n', '') in itens_ja_comprados]


with open('lista_de_compras_atualizada.txt', 'w') as lista_atualizada:
    lista_atualizada.writelines(itens_lista_compras)
```

Adicionando valores a um arquivo

Por fim, vamos utilizar o método write com o arquivo aberto no modo `'a'` para fazer uma adição a uma lista. Lembrando, que quando utilizamos o modo `'w'`, o arquivo original com o mesmo nome é sobrescrito. Já com o modo `'a'` (o a vem de *append*, acrescentar do inglês), o arquivo não é sobrescrito e permite que escrevamos mais dados ao arquivo selecionado.

```
itens_para_adicionar = ['farinha', 'fermento']

with open(pasta_atual / 'lista_de_compras.txt', 'a') as lista_compras:
    for item in itens_para_adicionar:
        lista_compras.write(f'\n{}'item)
```

Note que temos que adicionar o ‘\n’ para garantir a quebra de linha!

14. DESAFIO - Modificando arquivo html

Modificando um arquivo html

```
from pathlib import Path


item_remover = 'Passear com cachorro'


pasta_atual = Path(__file__).parent

with open(pasta_atual / 'desafio_textos' / 'view_lista.html') as html:
    linhas_html = html.readlines()

nova_linhas_lista = []
escrever_linha = True
for i, linha in enumerate(linhas_html):
    if i > len(linhas_html) - 3:
        break
    if item_remover in linhas_html[i + 2]:
        escrever_linha = False
    if escrever_linha:
        nova_linhas_lista.append(linha)
    if '</li>' in linha:
        escrever_linha = True

with open(pasta_atual / 'desafio_textos' / 'view_lista_atualizada.html', 'w') as html:
    html.writelines(nova_linhas_lista)
```

15. Lendo e Escrevendo Planilhas

Outra forma comum de armazenamento de dados é por planilhas. Quando falamos aqui em planilhas, estamos falando de arquivos com extensão para excel (xls e xlsx) ou arquivos CSV.

Vamos utilizar a biblioteca pandas para isso, já que pandas é a principal ferramenta para manipulação de tabelas em Python. Como pandas não é uma biblioteca padrão da linguagem, teremos que instalá-lo para utilização. Caso você não tenha pandas instalado no seu ambiente, basta utilizar pip da seguinte forma.

```
pip install pandas
```

Leitura de planilhas de Excel

Para ler planilhas, utilizaremos o método .read_excel:

```
import pandas as pd

tabela_clientes = pd.read_excel('clientes.xlsx')
print(tabela_clientes.head())
```

O método recebe diferentes argumentos, vamos explorar aqui os principais:

- `sheet_name` (default 0):
  - Nome da aba. Caso não seja passado, retorna a primeira aba.
  - Caso passe o valor 0, retorna a primeira aba, se passar o valor 1, retorna a segunda e assim por diante
  - Pode ser passado uma string com o nome da aba. Ex.: “nome_da_aba”
  - Pode ser passado uma lista, com nome ou números das abas. Ex.: [0, 1, “nome_da_aba”]. Isso retornará um dicionário de DataFrames como resultado.
- `header` (default 0):
  - Determina qual linha será utilizada como header da tabela. Caso seja, a primeira linha utilizar 0.
  - Caso não queira header, utilizar None.
- `names`:
  - Caso o header foi definido como None, você pode passar o nome das colunas como uma lista para definí-los
- `index_col` (default None):

  - Define qual coluna será tomada como índice do DataFrame. Deve ser passado como um valor inteiro, sendo que 0 representa a primeira coluna, 1 a segunda e assim por diante

- `thousands` (default None):
  - Para converter valores para numérico, quando esse valor está quebrado com pontos por exemplo. No caso, se temos na nossa tabelas valores numéricos escritos da seguinte forma: 1.000, teríamos que utilizar **thousands = '.'**

- `decimal` (default '.'):
  - Para converter valores para numérico valores decimais da tabela. No caso, se temos na nossa tabelas valores numéricos com vírgula escritos da seguinte forma: 10,800541, teríamos que utilizar **decimal = ','**

- `usecols` (default None):
  - Se None, retorna todas as colunas da planilha. Se "A:E", por exemplo, retorna apenas as colunas de A até E. Pode ser passado na forma de uma lista de ints também, por exemplo, [1, 2, 3, 4, 5] para pegar das colunas A até E.

- `skiprows`:
  - Números de linhas a serem ignoradas no início do arquivo

```
import pandas as pd

tabela_clientes = pd.read_excel('clientes.xlsx',
                                sheet_name='RS',
                                index_col=0,
                                decimal=',',
                                usecols=[0, 1, 2, 3],
                                skiprows = 1,
                                )
print(tabela_clientes.head())
```

Escrita de planilhas de Excel

Para ler planilhas, utilizaremos o método .to_excel:

```
import pandas as pd

tabela_clientes = pd.read_excel('clientes.xlsx')
tabela_clientes.to_excel('copia_clientes.xlsx')
```

O método recebe diferentes argumentos, vamos explorar aqui os principais:

- `sheet_name` (default 'Sheet1'):

    - Nome da aba que conterá a tabela

- `float_format`:
    - String de formato para floats. Por exemplo, float_format="%.2f" formatará de 0,1234 a 0,12.
- `columns`:
    - Colunas que serão escritas

Se você quiser escrever em mais de uma aba ao mesmo tempo, é necessário criar um objeto de escrita

```
df2 = df1.copy()
with pd.ExcelWriter('output.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet_name_1')
    df2.to_excel(writer, sheet_name='Sheet_name_2')
```

16. DESAFIO - Separando e Consolidando Planilhas

Separar diversas planilhas que estavam em abas diferentes, formatando elas antes e depois reuni-las novamente.

```
from pathlib import Path
import pandas as pd

pasta_atual = Path(__file__).parent


def separando_planilhas(caminho_planilha):
    tabela_clientes = pd.read_excel(caminho_planilha, sheet_name=None)

    for key, value in tabela_clientes.items():
        value.to_excel(pasta_atual / 'planilhas' / 'planilhas_separadas' / f'{key}.xlsx',
↪   index=False)

def consolidando_planilhas(caminho_separadas):
    with pd.ExcelWriter(pasta_atual / 'planilhas' / 'planilha_consolidada' / 'clientes.xlsx')
     ↪  as consolidada:
        for planilhas in caminho_separadas.glob('*.xlsx'):
            tabela_clientes = pd.read_excel(planilhas)
            tabela_clientes.to_excel(consolidada, sheet_name=planilhas.stem, index=False)
```

17. Lendo e Escrevendo Json

Json é um formato de dados muito utilizado para armazenamento e, principalmente, envio de dados. Grande maioria das API utilizadas na internet utilizarão JSons como formato padrão para armazenamento de dados.

Json significa JavaScript Object Notation e, apesar de ter começado com JavaScript, hoje é uma formatação independente e todas as linguagens passaram a utilizá-lo de alguma forma.

É muito utilizado para arquivos de configuração, por ter um formato fácil de parsear e de entender.

Você perceberá que o Json é muito similar a um dicionário em Python, o que já nos dá uma familiaridade com o tipo de dado.

Lendo e Escrevendo da memória

Digamos que recebemos um arquivo Json através de uma API. Para começar a utilizá-lo e manipulá-lo em Python, a melhor forma e decodificando ele utilizando o módulo json da biblioteca padrão. Ele transformará nosso arquivo json (que para o Python é lido como um string) para um dicionário, bem mais fácil de ser trabalhado. A leitura se daria da seguinte forma:

```
import json

data_json = '''
{
  "assinantes" : [
    {
      "nome": "Adriano Soares",
      "telefone": "51 99999999",
      "email": "adriano@email.com",
      "em_dia": true
    },
    {
      "nome": "Juliano faccioni",
      "telefone": "51 99999999",
      "email": "juliano@email.com",
      "em_dia": false
    }
    ],
  "data_extração": "2023/08/22"
}
'''

data = json.loads(data_json)
```

Já para retornarmos ao formato Json, caso queiramos reenviar essa informação através de uma API, por exemplo. Podemos fazer da seguinte forma:

```
data_json = json.dumps(data)
print(data_json)
```

Podemos utilizar o argumento indent para indentá-lo de uma forma que se torne mais fácil de ler:

```
data_json = json.dumps(data, indent=2)
print(data_json)
```

Lendo e escrevendo arquivos Json

Abrimos o arquivo com a função with da mesma forma que trabalhamos arquivos texto.

```
import json

with open('config.json') as f:
    data = json.load(f)

print(data)
print(type(data))
```

Para salvar o arquivo novamente em json, a seguinte forma é utilizada:

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

print(data)
print(type(data))
```

18. Lendo e Escrevendo Arquivos Pickle

Pickle é uma forma muito utilizada para serializar objetos de Python. O que queremos falar como isso?

Ocorre em Python de às vezes queremos salvar objetos no nosso disco para uso posterior. Por exemplo, criamos um modelo de Machine Learning e demorou diversas horar para o processador conseguir treinar o modelo. Digamos que eu não queira perder o trabalho já feito, para isso podemos salvar nosso modelo treinado em um arquivo pickle. Ele salvará o objeto exatamente no estado atual que ele se encontra naquele momento. Isso pode ser muito útil e poupar bastante tempo.

Com pickle, podemos salvar objetos simples como listas e dicionários. Para isso, utilizamos o módulo pickle da biblioteca padão de Python:

```
import pickle

minha_lista = [0, 1, 2]
meu_dict = {'a': 1, 'b': 2, 'c': 3}

with open('minha_lista.pickle', 'wb') as f:
    pickle.dump(minha_lista, f)

with open('meu_dict.pickle', 'wb') as f:
    pickle.dump(meu_dict, f)
```

O formato é similar a quando aprendemos a ler e escrever arquivos de texto. Isso por que pickle não passa de uma forma de escrever em bytes variáveis e objetos que estão namemória ram. Por isso, o modo utilizado é wb. Lembrando da aula de textos, o w é referente a escrita de um novo arquivo, já o b é referente a bytes, para informar que esse arquivo deve ser escrito como bytes, formando no final um arquivo que não é legível.

Para ler esses arquivos, o processo acaba sendo bem simples:

```
import pickle

with open('minha_lista.pickle', 'rb') as f:
    minha_lista = pickle.load(f)

with open('meu_dict.pickle', 'rb') as f:
    meu_dict = pickle.load(f)
```

Podemos explorar pickle com tipos de dados mais complexos como dataframes ou classes em geral.

```
import pickle

class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
```

```
    def quem_sou_eu(self):
        print(f'Eu sou {self.name} e tenho {self.idade} anos')

obj_pessoa = Pessoa('Carlos', 31)
obj_pessoa.quem_sou_eu()

with open('obj_pessoa.pickle', 'wb') as f:
    pickle.dump(obj_pessoa, f)
```

```
import pickle

with open('obj_pessoa.pickle', 'wb') as f:
    obj_pessoa = pickle.load(f)

obj_pessoa.quem_sou_eu()
```

19. Lendo e Escrevendo Arquivos XML

XML são similares a html. Os arquivos são estruturados a partir de tags que contem os dados armazenados. Segue abaixo um exemplo de arquivo xml.

```
<?xml version="1.0"?>
<catalogo>
   <livro id="bk101">
      <autor>Gambardella, Matthew</autor>
      <titulo>XML Developer's Guide</titulo>
      <genero>Computer</genero>
      <preco>44.95</preco>
      <data_publicacao>2000-10-01</data_publicacao>
      <descricao>An in-depth look at creating applications
      with XML.</descricao>
   </livro>
   <livro id="bk102">
      <autor>Ralls, Kim</autor>
      <titulo>Midnight Rain</titulo>
      <genero>Fantasy</genero>
      <preco>5.95</preco>
      <data_publicacao>2000-12-16</data_publicacao>
      <descricao>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</descricao>
   </livro>
</catalogo>
```

Arquivos xml são ainda mais poderosos que json, e formam a estrutura de grande parte dos sites. Muito do armazenamento de dados da web se baseia em arquivos xml ou similares. Por isso é importante já termos uma primeira visualização do que se trata esse dado, como podemos fazer manipulações simples nele e como escrevê-los novamente.

Essa apresentação será introdutória, até por ser uma estrutura de dados mais complexa e, consequentemente, o tratamento desse dado em Python é mais complexo.

Segue a forma de ler arquivos

```
import xml.dom.minidom
from pathlib import Path

pasta_atual = Path(__file__).parent
domtree = xml.dom.minidom.parse( 'livros.xml')

group = domtree.documentElement
livros = group.getElementsByTagName('livro')

for livro in livros:
    print(livro.getElementsByTagName('autor')[0].childNodes[0].nodeValue)
```

No caso, utilizamos `xml.dom` para abrir o arquivo xml. O que ele faz a transformação do arquivo em

elementos de DOM. Caso você não conheça, DOM é uma representação padrão de dados e compõe a estrutura e conteúdo de um documento Web. Através dele, podemos tanto representar arquivos HTML quanto XML.

Para aqueles que estão acostumado com DOM e programação em JavaScript, deve ter notado as familiaridades das funções que utilizamos. O método getElementsByTagName é padrão dentro de DOM, e o nome utilizado dentro de Python é exatamente o mesmo.

Não vamos abordar extensivamente os métodos de DOM, até por que não é escopo desse curso.

Deixo aqui mais uma breve explicação externa de DOM: clique aqui

Para escrever arquivos xml é simples:

```
with open(pasta_atual / 'xmls' / 'livros_copia.xml', 'w') as f:
    domtree.writexml(f)
```