Uma característica que nenhum de nossos programas teve até agora é a **persistência** de dados. Sempre que nossos programas eram executados, eles exigiam que os usuários digitassem todos os dados de entrada, e após exibir os dados de saída na tela, o programa era fechado e esses dados eram perdidos para sempre.

A persistência se dá através de **arquivos**: estruturas abstratas para armazenar dados em uma memória permanente, como o disco rígido, um *drive* USB ou um servidor *web*.

**1. Arquivos em Python**

O Python possui algumas funções prontas para manipular arquivos binários puros (onde, conhecendo a estrutura interna de qualquer formato, podemos salvar qualquer tipo de arquivo) e para manipular arquivos de texto (onde os binários são decodificados como *strings*).

Focaremos no básico de manipulação de arquivo de texto pois, na prática, quando formos trabalhar com arquivos mais complexos, é provável que usaremos bibliotecas específicas para lidar com eles, e elas já terão funções próprias para ler e salvar esses arquivos da maneira correta.

**1.1. Abrindo e fechando arquivos**

Podemos criar arquivos novos ou abrir arquivos já existentes utilizando a função open. Ela possui 2 argumentos: o caminho do arquivo e o modo de operação.

| **Modo** | **Símbolo** | **Descrição** |
| --- | --- | --- |
| *read* | r | lê um arquivo existente |
| *write* | w | cria um novo arquivo |
| *append* | a | abre um arquivo existente para adicionar informações ao seu final |
| *update* | + | ao ser combinado com outros modos, permite alteração de arquivo já existente (ex: r+ abre um arquivo existente e permite modificá-lo) |

Após abrirmos (ou criarmos) um arquivo, podemos realizar diversas operações. Ao final de todas elas, devemos **fechar** o  arquivo usando a função close. Essa etapa é importante por 2 motivos:

* 1. Se alteramos o arquivo mas não o fechamos, as alterações não serão salvas.
  2. Se esquecermos de fechar um arquivo, outros programas podem ter problemas de acesso a ele.

A função open retorna alguns dados de acesso ao arquivo que devem ser salvos em uma variável, para uso interno do Python.

**Atenção:** o modo 'w' sempre irá **criar um novo arquivo**. Caso você use esse modo para abrir um arquivo que já existe, o arquivo existente será substituído por um novo arquivo em branco, e seu conteúdo será perdido!

**1.2. Escrevendo arquivos**

Para entender melhor o open e o close, façamos um programinha que escreve algo em um arquivo. Além das duas funções que já vimos, também utilizaremos a função write, que escreve um texto em um arquivo. É quase como um print mais simples, mas ele aceita apenas uma *string*.

arquivo = open('ola.txt', 'w') # cria um arquivo ola.txt

arquivo.write('Olá mundo') # escreve "Olá mundo" no arquivo

arquivo.close() # fecha e salva o arquivo

Após executar o código acima, abra a pasta onde seu código-fonte está salvo. Note que apareceu o arquivo ola.txt dentro dela. Abra-o e verifique seu conteúdo.

Por padrão, arquivos serão escritos na mesma pasta onde está nosso código-fonte. Você pode passar caminhos completos caso prefira acessar outras pastas. Em vez de escrever por extenso o caminho de um arquivo, é recomendável o uso de uma biblioteca para construir os endereços, tanto para lidar melhor com possíveis erros quanto para garantir que nosso programa funcionará bem em computadores diferentes ou mesmo em sistemas operacionais diferentes. Um módulo bastante útil é o os.path, já instalado junto com o Python. [Aqui](https://linuxhint.com/os-path-module-python/) está um ótimo tutorial introdutório. Caso você precise se aprofundar, consulte a [documentação oficial](https://docs.python.org/pt-br/3.7/library/os.path.html).

**1.3. Lendo arquivos**

Para ler um arquivo existente, não basta usar o open para abri-lo. É necessário carregar seu conteúdo para uma *string*, de modo que possamos trabalhar com o texto da mesma forma que sempre trabalhamos. A função read faz o oposto da write: ela retorna o texto existente no arquivo.

Execute o código abaixo:

arquivo = open('ola.txt', 'r') #abre o arquivo já existente

conteudo = arquivo.read() #lê o conteúdo do arquivo e o salva na variável

print(conteudo)

arquivo.close()

Você pode estar se perguntando: nós aprendemos a criar um arquivo e escrever nele e aprendemos a ler o conteúdo de um arquivo já existente. Mas como realizamos modificações pontuais em um arquivo já existente? Uma forma seria utilizar as funções de manipulação de ponteiro de arquivo seek e tell. Você pode ler mais sobre elas [aqui](https://pynative.com/python-file-seek/). Porém, isso pode ser bastante trabalhoso, e na prática, quase sempre que formos realizar manipulação de arquivos mais complexos, utilizaremos bibliotecas que nos protegerão da manipulação direta do arquivo. Para manipular arquivos de texto simples, como estamos fazendo nesta aula, uma solução é sempre reescrever o arquivo inteiro: abra o arquivo em modo leitura (r) e carregue todo seu conteúdo em uma string. Faça as alterações desejadas na string, reabra o arquivo em modo de escrita (w) e escreva a string completa no arquivo.

**1.4. Gerenciador de contexto**

Uma forma alternativa e "mais segura" de trabalhar com arquivos é utilizando um *gerenciador de contextos*. O gerenciador de contextos é, de maneira resumida, um pequeno bloco de código que realiza algumas tarefas e tratamentos de erro de maneira automatizada.

Com ele, não precisamos nos preocupar em fechar o arquivo ao final da manipulação, pois ele faz isto automaticamente, ao final do bloco.

with open('ola.txt', 'r') as arquivo:

conteudo = arquivo.read()

print(conteudo.title())

# note que a linha abaixo gera um erro:

conteudo2 = arquivo.read()

# O erro é:

# ValueError: I/O operation on closed file. -> Operação de entrada/saída em arquivo fechado

No restante dos exemplos seguiremos utilizando a primeira forma que aprendemos para enfatizar que, por dentro do programa, há **sempre** uma abertura e um fechamento de arquivo. Mas recomendamos utilizar sempre que possível o gerenciador de contexto para garantir uma segurança maior.

Você pode ler mais sobre gerenciadores de contexto e aprender a criar o seu próprio neste ótimo artigo do [RealPython](https://realpython.com/python-with-statement/).

**2. Arquivos CSV**

Muitos dados interessantes ou importantes estão disponíveis na forma de tabela. A capacidade de manipular planilhas foi determinante no sucesso dos computadores pessoais, dada sua importância para empresas e indivíduos.

Aprenderemos a manipular dados utilizando um dos formatos de planilha mais amplamente utilizados na *web*: o formato CSV. Mas antes, como podemos representar tabelas em Python?

**2.1. Tabelas em Python**

Conforme já mencionamos, temos módulos prontos para realizar muitas tarefas para nós. Um dos módulos mais populares em Python é o *pandas*, que mesmo não vindo instalado por padrão é provavelmente o módulo mais usado para manipular planilhas. Porém, como este é um curso introdutório, convém entendermos um pouquinho de lógica de como manipular uma tabela para futuramente sermos capazes de trabalhar corretamente com os módulos prontos.

Uma das formas mais simples de se representar uma tabela em Python seria através de uma lista de listas. Nossa lista principal seria a tabela como um todo, e cada lista interna seria uma linha da tabela.

Para acessar um elemento individual, utilizamos 2 índices: o primeiro indica a lista interna (linha) e o segundo o elemento individual na lista (coluna). Para percorrer a tabela inteira, utilizamos 2 for aninhados: o mais externo fixa uma linha e o mais interno percorre cada elemento daquela linha. Execute o código abaixo e verifique o resultado mostrado na tela.

tabela = [['Aluno', 'Nota 1', 'Nota 2', 'Presenças'],

['Luke', 7, 9, 15],

['Han', 4, 7, 10],

['Leia', 9, 9, 16]]

print('Imprimindo cada elemento individual da tabela:')

for linha in tabela:

for elemento in linha:

print(elemento)

print('Imprimindo cada "linha" da tabela:')

for linha in tabela:

print(linha)

print('Imprimindo o elemento na linha 2, coluna 0:')

print(tabela[2][0])

**2.2. O formato CSV**

A sigla CSV significa *Comma-Separated Values*, ou "valores separados por vírgula". Este formato é uma forma padrão de representar tabelas usando arquivos de texto simples: cada elemento é separado por uma vírgula, e cada linha é separada por uma quebra de linha.

Na prática, nem sempre o padrão é seguido à risca: podemos utilizar outros símbolos para fazer a separação. Um bom motivo é o fato de a vírgula ser utilizada para representar casa decimal em algumas línguas, como a língua portuguesa. O importante é ser coerente: todos os elementos deverão ser separados pelo mesmo símbolo, e todas as linhas deverão ter o mesmo número de elementos.

Cole o texto abaixo em um editor de texto puro (como o Bloco de Notas, do Windows) e salve-o com a extensão .csv.

Aluno;Nota 1;Nota 2;Presenças

Luke;7;9;15

Han;4;7;10

Leia;9;9;16

Caso você tenha um editor de planilha instalado, como o Excel, é provável que o ícone representando o arquivo seja o ícone do editor de planilhas, e não de um arquivo de texto. Abra-o com seu editor de planilha e observe como ele interpreta corretamente os dados!

Devido ao fato de ser um formato aberto (ou seja, não é necessário pagar por propriedade intelectual para usar) e ser muito fácil de manipular, diversos programas diferentes possuem a opção de importar ou exportar dados em CSV, e diversas bases de dados na *web* oferecem a opção de baixar os dados neste formato.

**2.3. O módulo CSV em Python**

Devido à facilidade de trabalhar com arquivos CSV, com o que vimos sobre arquivos até o momento, já conseguimos facilmente escrever um programa que escreva uma planilha (representada como lista de listas) em um arquivo CSV. Da mesma forma, utilizando as funções que vimos de *strings*, conseguimos abrir um arquivo CSV e adequadamente reconhecer seus elementos (dica: lembre-se do método split).

Porém, como mencionamos antes, o Python possui muita coisa pronta, então não precisamos constantemente reinventar a roda. Existe um módulo chamado *csv* que já vem instalado com o Python. Ele já faz sozinho o serviço bruto de transformar nossa lista de listas em um texto separado por símbolos e vice-versa.

**2.3.1. Escrevendo um CSV**

Para escrever um CSV utilizando o módulo, precisamos ter nossos dados representados como uma lista de listas. Criaremos (ou abriremos) um arquivo usando o open, como já fizemos antes, e utilizaremos um *CSV writer* - uma estrutura que guardará as regrinhas para escrever nosso CSV. Execute o exemplo abaixo:

import csv

tabela = [['Aluno', 'Nota 1', 'Nota 2', 'Presenças'],

['Luke', 7, 9, 15],

['Han', 4, 7, 10],

['Leia', 9, 9, 16]]

# cria o arquivo CSV

arquivo = open('alunos.csv', 'w')

# definindo as regras do nosso CSV:

# ele será escrito no arquivo apontado pela variável 'arquivo'

# seus elementos serão delimitados (delimiter) pelo símbolo ';'

# suas linhas serão encerradas (lineterminator) por uma quebra de linha

escritor = csv.writer(arquivo, delimiter=';', lineterminator='\n')

# escreve uma lista de listas em formato CSV:

escritor.writerows(tabela)

# fecha e salva o arquivo

arquivo.close()

Após executar o programa acima, um arquivo *alunos.csv* deverá surgir na mesma pasta em que se encontra o seu código-fonte, e seu editor de planilhas provavelmente o reconhecerá com sucesso. Se você abrí-lo com um editor de texto puro, verá os dados separados por ;, assim como no arquivo que criamos anteriormente.

**2.3.2. Lendo um CSV**

O processo para ler o CSV é semelhante: utilizamos um *CSV reader*, com os mesmos parâmetros utilizados no *CSV writer*. A função csv.reader retorna uma estrutura iterável (ou seja, que pode ser percorrida com for) contendo cada linha já organizada como lista.

import csv

arquivo = open('alunos.csv', 'r')

planilha = csv.reader(arquivo, delimiter=';', lineterminator='\n')

for linha in planilha:

print(linha)

arquivo.close()

Note que a estrutura **não é** uma lista, mas um objeto iterável. Vejamos o que acontece se tentarmos imprimi-lo diretamente:

import csv

arquivo = open('alunos.csv', 'r')

planilha = csv.reader(arquivo, delimiter=';', lineterminator='\n')

print(planilha)

arquivo.close()

Saída na tela:

<\_csv.reader object at 0x032968B0>

Caso você precise de mais flexibilidade para trabalhar com a sua planilha - por exemplo, caso deseje editá-la, criar novas colunas etc, convém converter a estrutura para uma lista de verdade. É possível usar a função list no objeto para fazer a conversão:

import csv

arquivo = open('alunos.csv', 'r')

planilha = list(csv.reader(arquivo, delimiter=';', lineterminator='\n'))

arquivo.close()

print(planilha)

Saída na tela:

[['Aluno', 'Nota 1', 'Nota 2', 'Presenças'], ['Luke', '7', '9', '15'], ['Han', '4', '7', '10'], ['Leia', '9', '9', '16']]

Tudo que vem do teclado é considerado string e, por isso, frequentemente utilizamos coerção de tipos para interpretar estes dados como int ou float. O mesmo ocorre com documentos de texto. É comum salvarmos números em nossas planilhas, mas ao carregar o CSV, todos os campos serão strings. Você pode utilizar loops, compreensões de lista ou funções como o map para gerar uma tabela contendo números a partir de um arquivo CSV.

**3. Arquivos JSON**

JSON é uma sigla para *JavaScript Object Notation*. O *JavaScript* é uma linguagem muito utilizada em web, e assim como o Python, ela é uma linguagem orientada a objeto. Ocorre que a forma como objetos são representados nessa linguagem é bastante legível para seres humanos e fácil de decompor usando programação também.

Veja um exemplo de como podemos representar, por exemplo, um estudante em JavaScript:

{

nome: 'Mario',

modulo: 2,

media: 9.5

}

Parece familiar? É extremamente parecido com dicionários em Python. O Python possui um módulo já instalado chamado json que nos ajuda a realizar conversões entre uma *string* contendo um JSON e um dicionário.

**Atenção:** No caso do JSON faremos *exatamente* como nos dicionários em Python: as chaves deverão vir entre aspas.

Os valores de um JSON podem ser de vários tipos de dados: inteiros, reais, *strings*, booleanos, e até mesmo listas (representadas com colchetes) ou outros JSON/dicionários (representados por chaves). Por exemplo:

{

'escola':"Let's Code",

'cursos':[{'nome':'Python Pro', 'duracao':2},

{'nome':'Data Science', 'duracao':2},

{'nome':'Front-End', 'duracao':2}]

}

**3.1. JSON para dicionário**

O método loads recebe uma *string* contendo um JSON e retorna um dicionário, o que torna bastante fácil o acesso a informações individuais:

import json

jogador = '{"nome":"Mario","pontuacao":0}'

dicionario = json.loads(jogador)

print(dicionario['nome'])

print(dicionario['pontuacao'])

Saída na tela:

Mario

0

**3.2. Dicionário para JSON**

Já o método dumps recebe um dicionário e retorna uma *string* pronta para ser salva ou enviada como JSON:

import json

jogador = dict()

jogador['nome'] = 'Mario'

jogador['pontuacao'] = 0

string\_json = json.dumps(jogador)

print(string\_json)

Saida na tela:

{"nome": "Mario", "pontuacao": 0}

O dumps possui um parâmetro opcional indent que recebe um número inteiro. Isso fará com que a string gerada seja indentada, e o valor desse parâmetro determinará quantos espaços irão ao início de cada "nível".

 Sabendo realizar a conversão de dicionário para string e vice-versa, basta utilizar as técnicas de manipulação de arquivo de texto para ler a string do arquivo JSON, convertê-la para dicionário, fazer as manipulações desejadas, converter novamente para string e escrever no arquivo.