Programação Python

Aula 06: Strings e Arquivos Texto

Prof. Eduardo Corrêa Gonçalves

26/03/2021

Sumário

Tipos Básicos (1/3)

- **Tipos Básicos**
  - São os objetos mais primitivos, que armazenam valores simples.
  - Python possui quatro tipos básicos*:
    - **bool** : valores lógicos True/False
    - **int** : números inteiros
    - **float** : números reais
    - **str** : strings
  - Características:
    - São **objetos** (*detalhes no slide a seguir*)
    - São **imutáveis** (*detalhes na parte final da aula*)
    - A **string** além de ser tipo básico, é também um **container** e uma **sequência**.

** Há ainda o tipo **complex** para números complexos que não será coberto nessa aula*

Tipos Básicos (2/3)

- **Objetos**
  - Python é uma linguagem orientada a objetos.
  - **Tudo em Python é um objeto**, mesmo um valor com tipo básico.

```
>>> isinstance(1, object)
True

>>> isinstance(True, object)
True

>>> isinstance(3.14, object)
True

>>> isinstance("Olá ENCE !!!", object)
True

>>> isinstance([1,2,3], object)
True

>>> isinstance(("A","B"), object)
True
```

Tipos Básicos (3/3)

- **Classes e Objetos (2/2)**
  - O Python utiliza os termos **tipo** e **classe** como sinônimos.
  - Qualquer valor de tipo básico é tratado como um **objeto** que pode ser da **classe** bool, int, float ou str.
  - Nesta aula, abordaremos os objetos dos tipos básicos, com ênfase no tipo str.
    - O str é um tipo especial, pois além ser de básico é também um container e um sequência, possuindo muitos métodos.
    - Mas antes de falar do str, vamos descrever brevemente os tipos bool, int e float.

Classes bool, int e float (1/3)

- **classe bool**
  - Utilizada para representar valores lógicos: True e False.
  - Possui um construtor* **bool()** default que retorna False.
    - Ele quase nunca precisa ser usado, basta utilizar diretamente True e False.
    - O número 0 é avaliado como False; qualquer outro como True.
    - Containers vazios são avaliados como False; Caso contenham ao menos um elemento, True.
    - Uma aplicação importante desses conceitos é em testes lógicos de laços e estruturas de desvio.

```
>>> a = bool()
>>> a
False
>>> b = True
>>> c = False

>>> bool(1 < 2)
True

>>> bool(0); bool(1)
False
True

>>> bool([]); bool([1,2,3])
False
True
```

** Na Unidade 2 explicaremos o que é construtor*

Classes bool, int e float (2/3)

- **classe int**
  - Utilizada para manipular números **inteiros** de **magnitude arbitrária**.
    - Não existem subtipos como *tinyint* ou *longint*.
  - O construtor **int()** retorna 0 por default.
    - Se passo um float, o valor será truncado.
    - Se passo um outro tipo, converte para inteiro se possível ou retorna erro caso não seja possível

```
>>> a = 1024
>>> b = -16
>>>
>>> c = 2 ** 64
>>> c
18446744073709551616

>>> int()
0

>>> int(3.14); int(5.99); int(-3.9)
3
5
-3

>>> int('2048')
2048
>>> int('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base
10: 'hello'
```

Classes bool, int e float (3/3)

- **classe float**
  - Tipo ponto flutuante, utilizado para manipular números **reais**.
    - É possível utilizar notação científica.

```
>>> a = 3.14; b = 2.0;
>>> c = 999.9999999999
>>> d = 6.022e23
>>> d
6.022e+23
```

  - O construtor **float()** retorna 0.0 por default.

```
>>> float()
0.0
```

    - Divisão com “ / ” sempre retorna float.

```
>>> 10 / 2
5.0
```

    - Se passo um parâmetro, converte para float se possível ou retorna erro caso não seja possível

```
>>> float(256)
256.0

>>> float('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float:
'hello'
```

Classe str - String (1/9)

- **Definição**
  - Uma string em Python é uma **sequência de caracteres**.
    - Por isso, mesmo que str seja um tipo básico, qualquer string pode ser manipulada como uma **tupla** de caracteres

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
E	s	t	a	t	í	s	t	i	c	a

  - Características:
    - **Imutável** – nenhum caractere pode ser alterado.
    - **Sequência** – elementos possuem ordem determinada.
    - **Iterável** – capaz de retornar seus elementos (caracteres) um por vez em um laço.

Classe str - String (2/9)

- **Criando Strings**
  - Devemos especificar um texto entre aspas simples ou duplas.
  - O construtor **str()** pode ser usado para converter um número ou outro tipo em String.

```
escritor = 'Jorge Amado’
curso = "Programação Avançada"
x = str(1000)    #converte de int para str
n = str(90.99)    #converte de float para str
string_vazia = str()    #cria string vazia
```

Classe str - String (3/9)

- **Operações básicas**:
  - Recuperar caractere do índice *i*
  - Verificar se caractere ou substring pertence à string
  - Fatiar (obter uma substring)
  - Iterar (percorrer todos os caracteres *em ordem*)
  - Concatenar strings
  - Comparar duas strings
  - Converter para maiúsculo / minúsculo
  - Trocar uma substring por outra (*replace*).
  - Remover espaços em branco à direita ou esquerda.
  - Contar número de ocorrências de uma substring *s*
  - Verificar se uma substring *s* está contida na string.
  - Split
  - Remover pontuações e acentos.
  - ...

Classe str - String (4/9)

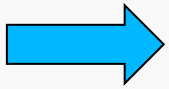
- **Operações – indexação e fatiamento (iguais à Tupla)**

```
s = "John Lennon"
s[0]                    # 'J'
s[:4]                   # 'John'
s[::-1]                 # 'nonneL nhoJ'
s + " – Imagine"        # "John Lennon – Imagine" (concatenação)
"L" in s                # True
"Lennon" in s           # True – busca tanto caractere, como substring inteira!
"lennon" in s           # False – diferencia maiúsculas e minúsculas
type(s)                 # <class 'str'>
len(s)                  # 11 – tamanho de s (número de caracteres)

s[0] = "j"              # não pode modificar nenhum caractere
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Classe str - String (5/9)

- **Iterando**

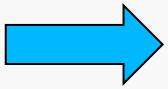
```
nome = "John"

for letra in nome:
    print(letra)
```



```
J
o
h
n
```

- **Iterando com base nos índices**

```
for k in range(len(nome)):
    print("elemento {} = {} ".format(k, nome[k]))
```



```
elemento 0 = J
elemento 1 = o
elemento 2 = h
elemento 3 = n
```

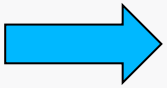
Classe str - String (6/9)

- **Comparação de strings**
  - As comparações são feitas com base no **código interno** (Unicode) de cada caractere.
    - A comparação é feita **letra por letra**.
    - Duas strings são **iguais** apenas se armazenam a **mesma palavra, escrita de maneira idêntica** (incluindo maiúsculo/minúsculo e acentos).
    - Letras maiúsculas possuem códigos menores do que as minúsculas ... Números têm códigos menores que qualquer letra ... letras acentuadas código maior do que as não acentuadas, ... (*veja a tabela Unicode*).

```
p1 = 'áaa'
p2 = 'aaa'
p3 = 'AAA'
print(p1==p2)  #False
print(p2 < p1)  #True
print(p2 < p3)  #False
```

Classe str - String (7/9)

- **Comparação de strings**
  - Em Português torna-se útil **converter para minúsculo** (ou maiúsculo) e **remover acentos** antes de comparar strings.
    - Porém, só entraremos em detalhe sobre este tema na aula sobre Data Wrangling com a biblioteca ‘pandas’.

"Acentuação"  "acentuacao"

Classe str - String (8/9)

- **Métodos disponíveis para strings (*apenas os principais*)**
  - s.**lower()** : retorna um clone de *s* com todas letras convertidas para minúsculo.
  - s.**upper()** : retorna um clone de *s* com todas letras convertidas para maiúsculo.
  - s.**find(*sub, início, fim*)**: verifica se *sub* ocorre na string *s* ou dentro de algum trecho específico caso *início* e *fim* tenham sido definidos. Se *sub* é encontrada, retorna o índice da primeira ocorrência. Senão retorna -1;
  - s.**rfind(*sub, início, fim*)**: igual ao find(), mas verifica de trás pra frente.
  - s.**endswith(*suf, início, fim*)**: verifica se *s* ou uma fatia de *s* termina com o sufixo *suf*.
  - s.**replace(*sub_ant, sub_nova, max*)**: retorna uma cópia de *s* com as ocorrências da substring *sub_ant* substituídas por *sub_nova*. O parâmetro *max* pode ser utilizado para determinar o número máximo de trocas que serão realizadas.
  - s.**count(*sub, início, fim*)**: conta número de ocorrências da substring *sub* em *s.*
  - s.**strip**(), s.**lstrip()**, s.**rstrip()**: retorna cópia de *s* sem espaços em branco à esquerda e direita, apenas à esquerda ou apenas à direita, respectivamente.
  - s.**split(*d, max*)**: cria uma lista de strings a partir de *s*, de acordo com o delimitador *d* (se este não tiver sido fornecido, usa espaço em branco). O parâmetro *max* pode ser utilizado para determinar o número máximo de elementos da lista.
  - ...

Classe str - String (9/9)

- **Utilizando os métodos**

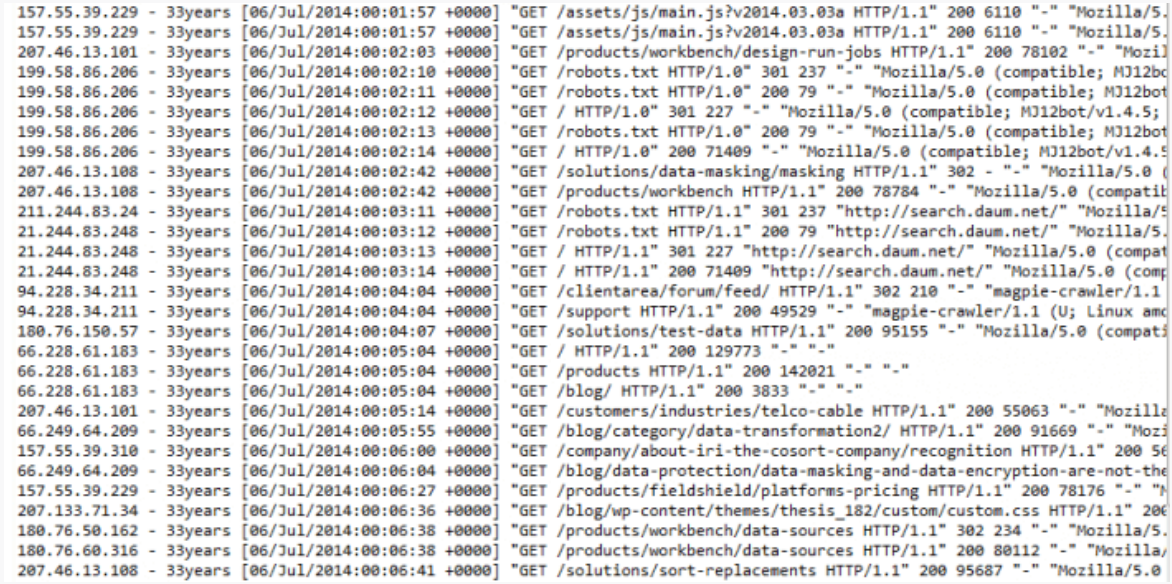
```
p1 = 'Lagarto Teiú'
p1.count('a')              #2 (possui dois “a” minúsculos)
p1.endswith('rto',0,7)     #True - os 3 últimos caracteres entre 0 e 6 são ‘r’,’t’,’o’
p1.find('a')               # 1 (primeira ocorrência de “a” é na posição 1)
p1.find('abc')             # -1 (substring “abc” não existe em p1)

p1.upper()                 # ‘LAGARTO TEIÚ’
p1.lower()                 # ‘lagarto teiú’

p1.replace('a', 'o')               # Logorto Teiú
p1.replace('Teiú', 'Verde’)        # Lagarto Verde

p1.split()                 #['Lagarto', 'Teiú’]

" capivara    ".strip()    # ‘capivara’
```

Arquivos Texto (1/12)

- Um arquivo é um conjunto de contíguo de bytes relacionados, mantido em algum dispositivo de armazenamento permanente (HD, pen drive, cartão de memória, etc.).

- Como cientista de dados, uma das tarefas mais comuns que você realizará será ler e escrever (gravar) **arquivos** no formato **texto**.

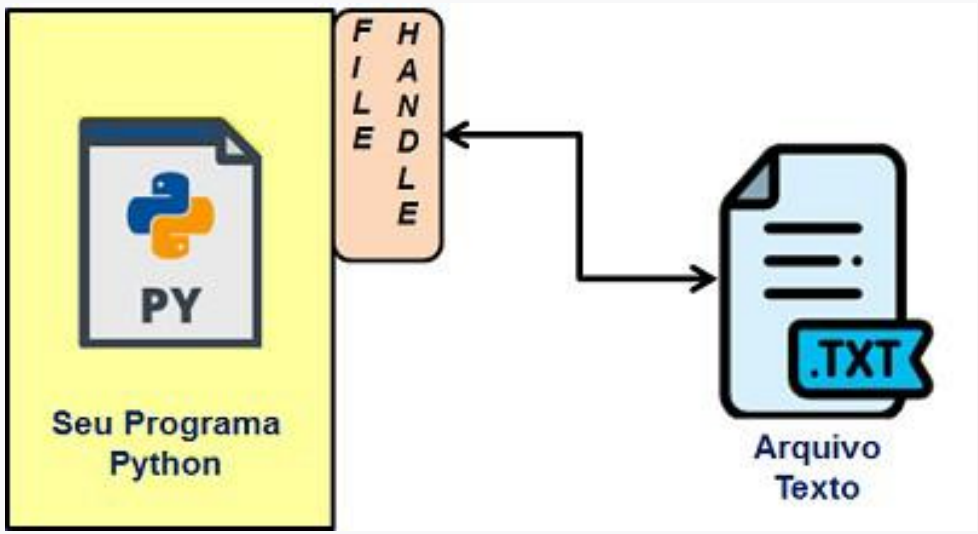
- **Exemplos**:
  - Arquivos CSV;
  - Logs ----------→
  - Arquivos XML*;
  - Arquivos JSON*;
  - *etc.*



* Estes tópicos serão abordados em outra aula

Arquivos Texto (2/12)

- **Função open()**
  - No Python padrão, é necessário usar a função **open()** para comandar a **abertura** de um arquivo.
  - **Abrir** significa pedir ao SO para **encontrar o endereço** de localização do arquivo no HD, pen drive, etc.
  - Ao encontrar o endereço do arquivo, o SO retornará um **file handle** para o programa.
    - O file handle **não** é a mesma coisa que conteúdo (dados) do arquivo.
    - Na verdade, é uma ferramenta que permite ao programador "manejar" os dados do arquivo.



Arquivos Texto (3/12)

- **Função open() - Importância**
  - Trabalhar com arquivos via open() / file handle **não** é tão “confortável” como trabalhar com um Data Frame do pacote pandas, por exemplo.
  - No entanto, **é importante** saber lidar com a função open() pelos seguintes motivos:
    - É uma ferramenta **eficiente** para realizar o **acesso sequencial** (linha por linha). Muitas vezes, esta é a única opção viável para lidar com bases muito grandes.
    - É a ferramenta mais simples para lidar com **arquivos separados por colunas** (formato raro hoje em dia, mas ainda usado).
    - Trata-se de uma função do **Python padrão**. Ou seja: permite com que você trabalhe com arquivos sem precisar instalar ou importar nenhum pacote adicional.

Arquivos Texto (4/12)

- **Função open() – Sintaxe**

  f = **open**(nome_arq, modo)

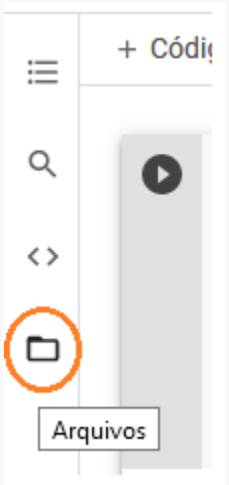
  - **nome_arq**: nome do arquivo (incluindo especificação do diretório). Único parâmetro obrigatório.
  - **modo**:
    - **'r'**: abre o arquivo para leitura (default)
    - **'w'**: abre o arquivo para escrita, eliminando a versão anterior.
    - **'a'**: abre o arquivo no modo append. Se o arquivo existir, as novas linhas serão inseridas no final.
      - **'rb', 'wb', 'ab'** nos permitem trabalhar com arquivos binários, em vez de arquivos texto.
  - Por exemplo, comando abaixo, abre para leitura o arquivo 'weather.csv", localizado na pasta "c:\bases"
    - f = **open**('C:/bases/weather.csv')

Arquivos Texto (5/12)

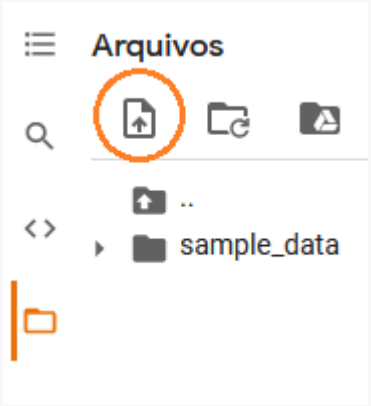
- **Google Colab – IMPORTANTE !!!**

  - Como os notebooks Colab ficam hospedados nos servidores do Google, eles não podem acessar diretamente o HD da sua máquina.
  - Por isso, torna-se necessário fazer o upload dos arquivos que você irá utilizar.

1. Clique em Arquivos (no lado esquerdo)



2. Clique em Upload e envie o arquivo



- *Detalhes sobre outras formas em: https://neptune.ai/blog/google-colab-dealing-with-files*

Arquivos Texto (6/12)

- **Função open() – Uso básico**

```
f = open('C:/bases/weather.csv')
```

  - E o que a função retorna?
    - **Não** é o conteúdo do arquivo!
    - Mas sim, um objeto da classe _io.TextIOWrapper.
      - Essa classe oferece métodos para manipular arquivos.

```
>>> f
<_io.TextIOWrapper name='C:/CursoPython/PRODUTOS.txt' mode='r' encoding='cp1252'>
>>>
>>> type(f)
<class '_io.TextIOWrapper'>
```

Arquivos Texto (7/12)

- **Acesso Sequencial**
  - A forma mais comum de acessar arquivos texto no Python padrão é **processando linha por linha**, sequencialmente.
  - Neste modelo, o Python permite com que um arquivo seja visto como uma ED, mais especificamente um **container de linhas**.
    - É como se fosse uma fila de linhas. Podemos "desenfileirá-las" uma por uma na ordem em que estão dispostas no arquivo.
    - Muito **eficiente** em termos de **ocupação de memória**. A cada iteração, apenas uma linha reside na memória.

arq_colunas.txt

```
1001aaaaa
1002bbbbb
1003ccccc
1004ddddd
1005eeeee
```

```
f = open('ARQ_COLUNAS.txt')
for linha in f:
    print(linha)

f.close()

>>>
1001aaaaa

1002bbbbb

1003ccccc

1004ddddd

1005eeeee
```

Arquivos Texto (8/12)

- **Acesso Sequencial – Sintaxe:**
  - A sintaxe básica para ler um arquivo sequencialmente é muito simples:

```
for linha in f:
    comandos
```

  - A cada iteração do **for**, a próxima linha será armazenada como uma **string** na variável “linha”.
  - Veja que incialmente, precisamos abrir o arquivo com o método open().
  - No final, fechá-lo com o método close().
    - Na operação de leitura isso não é obrigatório, mas é uma boa prática.

arq_colunas.txt

```
1001aaaaa
1002bbbbb
1003ccccc
1004ddddd
1005eeeee
```

```
f = open('ARQ_COLUNAS.txt')
for linha in f:
    print(linha)

f.close()

>>>
1001aaaaa

1002bbbbb

1003ccccc

1004ddddd

1005eeeee
```

Arquivos Texto (9/12)

- **Problema: o caractere “\n”**
  - Ao executar o exemplo, você deve ter percebido que uma linha em branco foi impressa depois de cada print().
  - Isto ocorreu porque o último caractere de uma linha do arquivo é na verdade “invisível”.
  - Ele se chama **fim de linha** ou **newline** e é representado por “**\n**” nas linguagens de programação.
    - Na verdade o fim de linha no Windows é representado por dois caracteres invisíveis (CR + LF). No Windows e Mac, apenas um (LF).
    - Para simplificar, referenciaremos o fim de linha sempre como \n

```
f = open('ARQ_COLUNAS.txt')
for linha in f:
    print(linha)

f.close()

>>>
1001aaaaa

1002bbbbb

1003ccccc

1004ddddd

1005eeeee
```

Arquivos Texto (10/12)

- **Problema: o caractere “\n”**
  - Veja que se mandamos imprimir o comprimento da linha, o valor retornado é 10, embora a gente só enxergue 9 caracteres.

```
f = open('ARQ_COLUNAS.txt')
for linha in f:
    print("comprimento = ", len(linha))
    print(linha)

f.close()

>>>
comprimento =  10
1001aaaaa

comprimento =  10
1002bbbbb

comprimento =  10
1003ccccc

comprimento =  10
1004ddddd

comprimento =  10
1005eeeee
```

Arquivos Texto (11/12)

- **Solução: usar os métodos rstrip ou len()**
  - Para descartar o “\n” após ler uma linha, você pode utilizar:
  - linha.**rstrip()**:
    - Este método remove espaços em branco à direita e também o “\n”
  - linha[:**len(**linha**)**-1]:
    - Fatia a linha do caractere 0 até o penúltimo caractere.
    - Ou seja, remove o “\n”, pois ele é o último caractere.

```
f = open('ARQ_COLUNAS.txt')
for linha in f:
    print(linha.rstrip())

f.close()

>>>
1001aaaaa
1002bbbbb
1003ccccc
1004ddddd
1005eeeee
```

Arquivos Texto (12/12)

- **Acesso Sequencial - Resumo**
  - É o modo de acesso a arquivos mais usado no Python padrão.
  - Recupera as linhas do arquivo, **uma por uma**, da primeira à ultima, em um laço **for**.
  - Cada linha é sempre lida para uma variável **string**.
  - Só permite avançar as linhas, **nunca recuar**.
  - **Eficiente**, especialmente na questão da ocupação de memória.

  - Nos próximos slides, mostraremos a **receita** para processar sequencialmente bases de dados texto em dois diferentes formatos:
    - Arquivo separado por colunas;
    - Arquivo separado por delimitador;

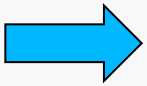
  - Para rodar os exemplos, baixe as bases de dados no repositório e as coloque na mesma pasta dos programas.

BDs Texto (1/4)

- **I. Leitura de Arquivo Separado por Colunas – Tamanho Fixo**
  - Em “arq_colunas.txt” temos duas variáveis:
    - Uma delas numérica, da coluna 0 a 3.
    - A outra é categórica, da coluna 4 a 8.
    - Solução: Usar o **fatiamento** para separar as variáveis.

arq_colunas.txt

```
1001aaaaa
1002bbbbb
1003ccccc
1004ddddd
1005eeeee
```



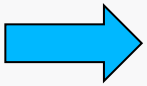
```
nomeArq = 'ARQ_COLUNAS.txt'
f = open(nomeArq)
for linha in f:
    v1 = int(linha[:4])  #usei int p/ converter v1 p/ inteiro
    v2 = linha[4:9]
    print(v1, v2)

>>>
1001 aaaaa
1002 bbbbb
1003 ccccc
1004 ddddd
1005 eeeee
```

BDs Texto (2/4)

- **II. Leitura de Arquivo Separado por Colunas – Tamanho Variável**
  - Em “produtos.txt”, também temos duas variáveis:
    - Código do produto, da coluna 0 a 3
    - Nome, da coluna 4 até a última coluna (varia a cada linha).
    - Solução: usar **len()** para capturar o tamanho de cada linha.

produtos.txt

```
1001Leite
1002Biscoito
1003Café
1004Torradas
1005Chá
```



```
nomeArq = 'PRODUTOS.txt'
f = open(nomeArq)
for linha in f:
    codigo = linha[:4]
    nome = linha[4:len(linha)-1]
    print(codigo, nome)
>>>
1001 Leite
1002 Biscoito
1003 Café
1004 Torradas
1005 Chá
```

BDs Texto (3/4)

- **III. Leitura de Arquivo Separado por Delimitador (1/2)**
  - No arquivo “ceps.csv”
    - A primeira linha é o cabeçalho e as seguintes contém as observações.
    - 3 variáveis separadas por vírgula.
    - Solução: usar o método **split()**, que divide a string em uma lista, bastando especificar o **delimitador**, neste caso, a vírgula “,”.

ceps.csv

```
cep_ini, cep_fim, nome_uf
20000000,28999999,Rio de Janeiro
29000000,29999999,Espírito Santo
30000000,39999999,Minas Gerais
01000000,19999999,São Paulo
```

BDs Texto (4/4)

- **III. Leitura de Arquivo Separado por Delimitador (2/2)**

```
nomeArq = 'CEPS.csv'
f = open(nomeArq)

aux=0  #auxiliar para permitir que cabeçalho seja ignorado
for linha in f:
    if (aux > 0): #ignora a linha de cabeçalho
        linha = linha.rstrip() #remove o tremendamente chato do "/n"
        lstPalavras = linha.split(",")
        cep_ini = lstPalavras[0]
        cep_fim = lstPalavras[1]
        uf = lstPalavras[2]
        print(uf + " -> CEPS de " + cep_ini + " a " + cep_fim)
    aux=aux+1

>>>
Rio de Janeiro -> CEPS de 20000000 a 28999999
Espírito Santo -> CEPS de 29000000 a 29999999
Minas Gerais -> CEPS de 30000000 a 39999999
São Paulo -> CEPS de 01000000 a 19999999
```

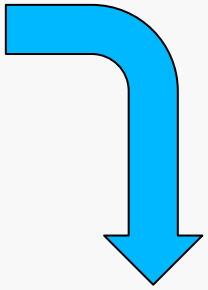
Gravando Arquivos (1/3)

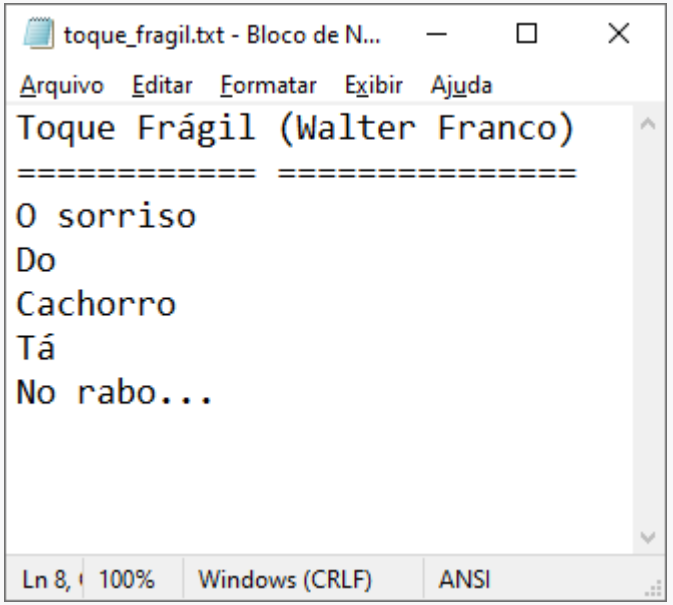
- **I. Gravando um Arquivo (1/2)**
  - Você deve abrir o arquivo utilizando o modo "w".
    - **CUIDADO**: caso um arquivo com mesmo nome exista, ele será destruído!
  - Quando a gravação terminar, é necessário utilizar a função **close()** para fechar o arquivo.
  - A função **write()** é utilizada para gravar uma linha.
    - **Obs.**: sempre que você quiser uma quebra de linha, deverá especificar explicitamente o "/n"

Gravando Arquivos (2/3)

- **I. Gravando um Arquivo (2/2)**

```
fout = open('toque_fragil.txt', 'w')
msg1 = "O sorriso\n"
msg2 = "Do\n"
msg3 = "Cachorro\n"
msg4 = "Tá\n"
msg5 = "No rabo...\n"

fout.write("Toque Frágil (Walter Franco)\n")
fout.write("=========== ==============")
fout.write("\n")
fout.write(msg1 + msg2 + msg3 + msg4 + msg5)
fout.close()
```

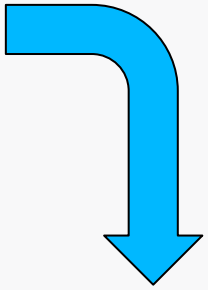


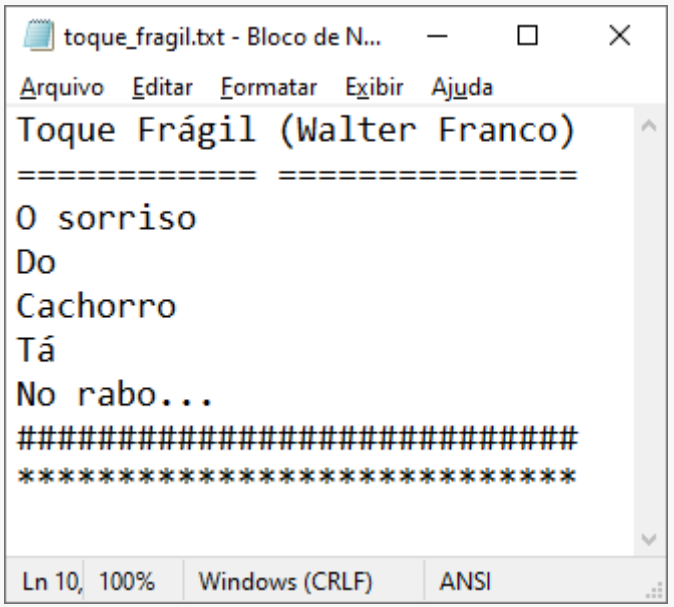


Gravando Arquivos (3/3)

- **II. Inserindo Linhas no Final de um Arquivo Existente**
  - Você deve abrir o arquivo utilizando o modo “a” (append).

```
fout = open('C:/CursoPython/toque_fragil.txt', 'a')
fout.write("##############################\n")
fout.write("******************************\n")
fout.close()
```





Resumo

- **Arquivos Texto – Processamento Sequencial**
  - Com o uso do Python padrão, podemos processar arquivos texto sequencialmente de forma simples e eficiente.
  - Neste modo de processamento, um arquivo é visto como um container, similar a uma fila.
    - Cada elemento do container é uma linha.
    - As linhas são acessadas uma por vez, a partir da primeira.
      - No acesso sequencial, só podemos avançar linhas, nunca retroceder.
    - Cada linha é encerrada por um newline ou “\n”, que é invisível.

Tarefas

- DOJO1: ANSI *versus* UTF-8

- DOJO2: Processando uma base de dados baixada do UCI

Referências

- Corrêa, E. (2020). “Meu Primeiro Livro de Python”. v2.0.0, edubd. (*capítulo 4*).
  - Disponível em: https://github.com/edubd/meu_primeiro_livro_de_python

- Python 3 Tutorial – File Management https://www.python-course.eu/python3_file_management.php