



Manipulação de dados em arquivos

Prof. Frederico Tosta de Oliveira

Prof. Kleber de Aguiar

Descrição

Formas de armazenamento e recuperação de dados em arquivos e diretórios, utilizando a linguagem de programação Python.

Propósito

Compreender os passos necessários para manipulação de arquivos e strings, utilizando boas práticas e tratamento de exceção, para garantir o correto funcionamento do programa.

Preparação

Antes de iniciar o conteúdo, sugerimos que tenha uma versão do software Python instalado em seu computador. Você pode baixá-lo no site oficial do Python.

Para programar os exemplos apresentados, utilizamos a IDE Pycharm Community, encontrada no site oficial da JetBrains. Baixe os [códigos fonte Python](#), para lhe auxiliar na realização das atividades.

Objetivos

Módulo 1

Funções de manipulação de arquivos

Identificar as funções de manipulação de arquivos.

Funções de manipulação de strings

Reconhecer as funções de manipulação de strings.

Tratamento de exceções e outras operações

Descrever as exceções na manipulação de arquivos e outras operações.



Introdução

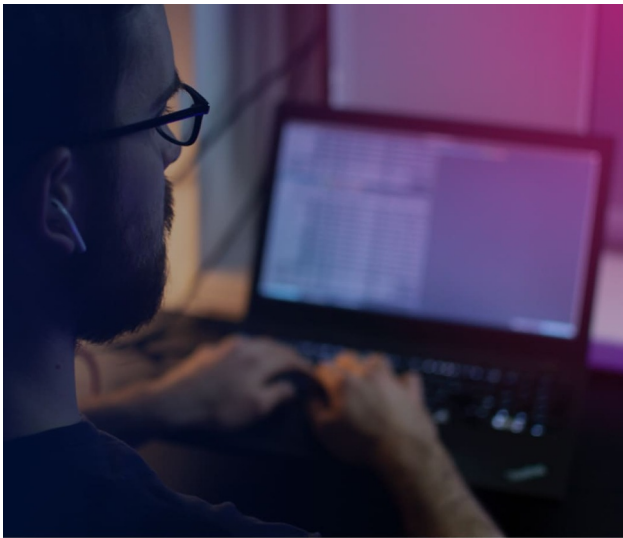
Uma das razões pelas quais o Python se popularizou foi a facilidade de programar utilizando essa linguagem.

A manipulação de arquivos em Python não é diferente. Os criadores se preocuparam em disponibilizar aos desenvolvedores o que realmente importa!

Com nome de métodos intuitivos, como *read* (ler) e *write* (escrever), e tratamentos de exceções específicas para cada problema, o Python disponibiliza uma simples e poderosa forma de trabalhar com arquivos. Neste conteúdo, veremos como manipulá-los de forma correta, garantindo que o programa rode sem problemas.

É muito comum ajustarmos e prepararmos um texto antes de incluí-lo em um arquivo. Ajustes como remover espaço em branco, colocar todas as letras maiúsculas, substituir palavras e adicionar conteúdo de variável são alguns exemplos. Esses ajustes podem ser realizados a partir de métodos de manipulação de strings, que também serão tratados adiante.





1 - Funções de manipulação de arquivos

Ao final deste módulo, você será capaz de identificar as funções de manipulação de arquivos.

Operações básicas

Conceitos

Abrindo um arquivo

Veja as operações básicas de manipulação de arquivos:



Abrir



Fechar



Ler



Escrever

A primeira operação que precisamos realizar, independentemente se vamos ler o conteúdo de um arquivo ou adicionar um conteúdo, é **abrir o arquivo**.

Para abrir um arquivo, o Python disponibiliza a função interna chamada `open`. Essa função está disponível globalmente, ou seja, não é preciso importá-la.

A função `open` retorna um objeto do tipo arquivo. Sua forma mais simples de utilização tem a seguinte sintaxe:

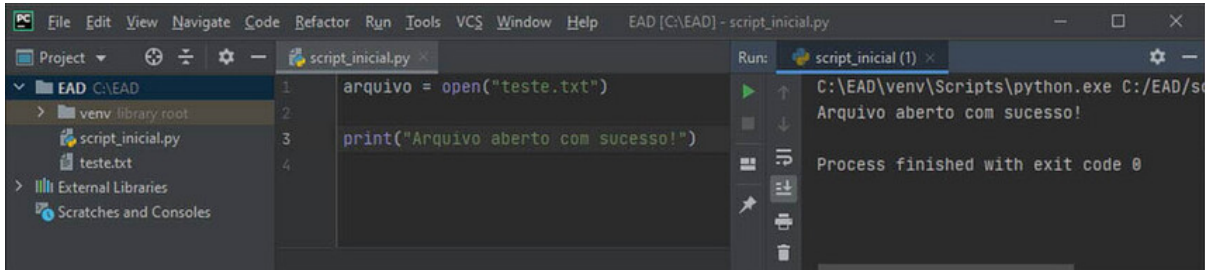
Python



```
1 arquivo = open (caminho)
```

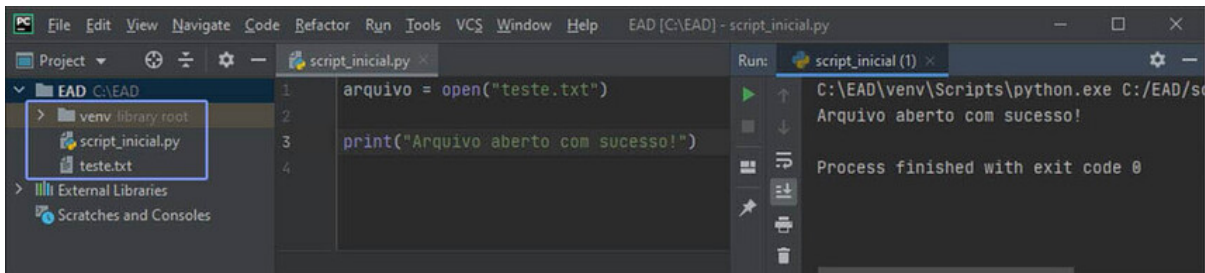
Utilizamos a função *open* com o parâmetro caminho. Esse parâmetro é uma string que representa a localização do arquivo no sistema de arquivos.

Veja como é fácil abrir um arquivo:



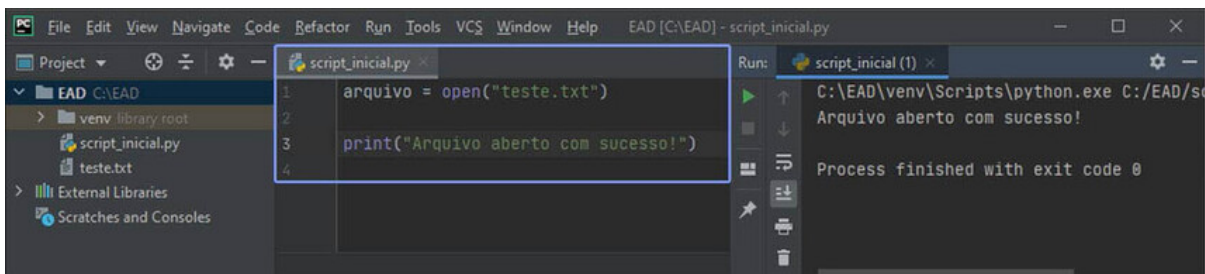
Abertura do arquivo.

Temos, inicialmente, o script inicial e sua saída.



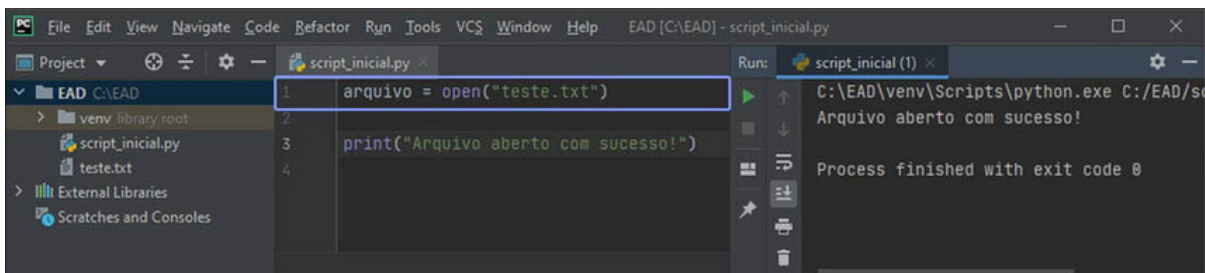
Arquivos script_inicial.py e teste.txt.

Temos, à esquerda da imagem, a árvore de diretório, onde verificamos a existência dos arquivos script_inicial.py e teste.txt, ambos no mesmo diretório EAD.



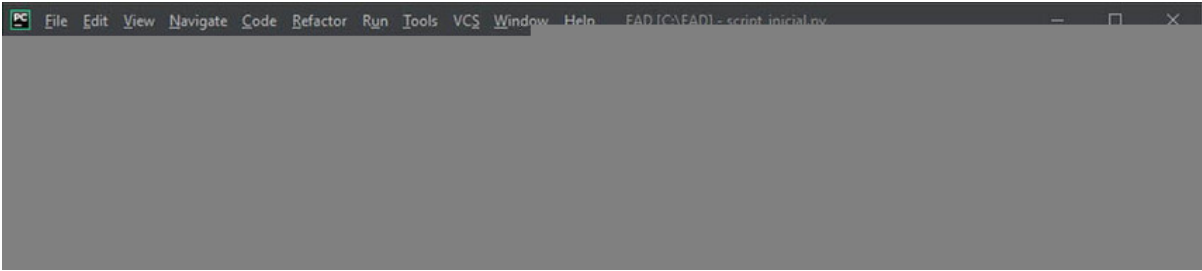
Código do script e a saída de controle.

Temos, ao centro, o código do nosso script e, à direita, a saída do console.



A função *open* para a abertura do arquivo teste.txt.

Utilizamos, na linha 1 do script, a função *open* para abrir o arquivo teste.txt. Isso já é suficiente para termos um objeto do tipo arquivo e começar a manipulá-lo.



Impressão da frase “Arquivo aberto com sucesso!”.

Imprimimos, na linha 3, a frase “Arquivo aberto com sucesso!” apenas para verificar se o programa foi executado sem problemas.

Nesse exemplo, o caminho utilizado para abrir o arquivo foi “teste.txt”, pois o script e o arquivo que abrimos estão no mesmo diretório. Porém, não precisamos nos limitar a manter os arquivos e scripts no mesmo diretório.

Veja como o Python trata o acesso aos arquivos a seguir. O caminho de um arquivo pode ser classificado em dois tipos:

Absoluto

É a referência completa para se encontrar um arquivo ou diretório. Ele deve começar com uma barra (/) ou o rótulo do drive (C:, D: ...).

Exemplo:

- open(“C:\Downloads\arquivo.txt”) – utilizado em ambientes MS Windows.
- open(“/home/usuario/arquivo.txt”) – utilizado em ambientes Linux.

Relativo

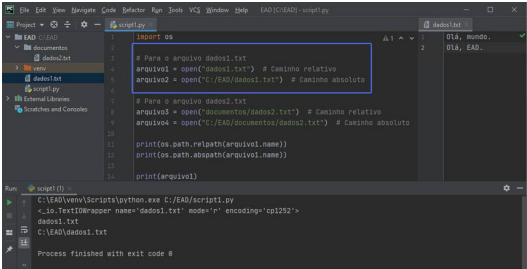
É a referência para se encontrar um arquivo ou diretório a partir de outro diretório. Normalmente, a partir do diretório de onde o script está.

Exemplo:

- open(“arquivo.txt”), para os casos em que o arquivo está no mesmo diretório do **script**.
- open(“../arquivo.txt”), para os casos em que o arquivo está no diretório acima do **script**.

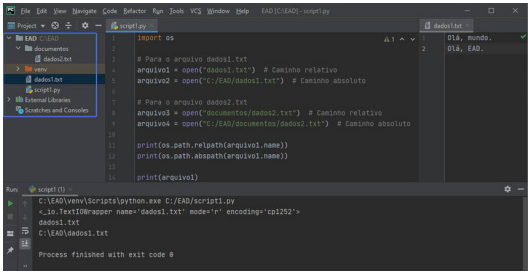
Vamos criar um script que ilustra as diferentes formas de referenciar um arquivo com caminhos absolutos e relativos.

No exemplo, a seguir, alteramos um pouco a forma de exibir o conteúdo:



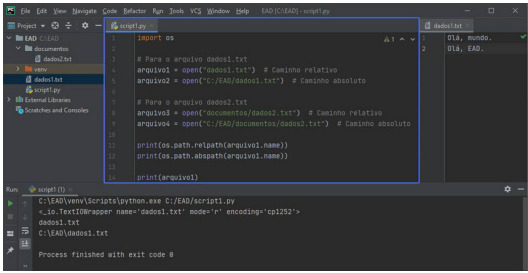
Script 1, sua saída e o arquivo dados1.txt.

Temos, inicialmente, o **script1**, sua saída e arquivo dados1.txt.



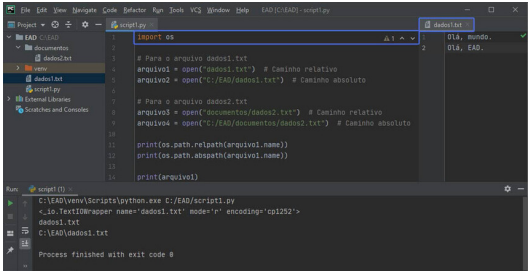
Árvore de diretórios.

Temos, **à esquerda**, nossa árvore de diretórios.



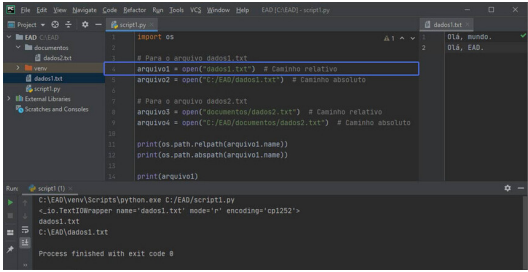
O script1.py.

Temos, **ao centro**, o script1.py.



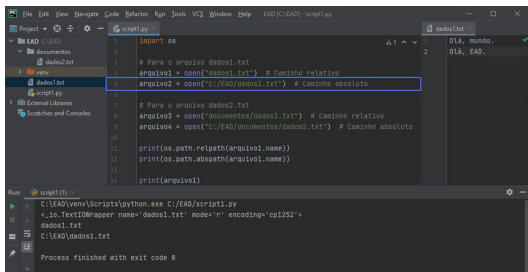
O arquivo dados.txt e saída do console do Python.

Temos, **à direita**, o arquivo dados.txt, e, abaixo, a saída do console do Python. Na linha 1 do script1.py, importamos o módulo os do Python.



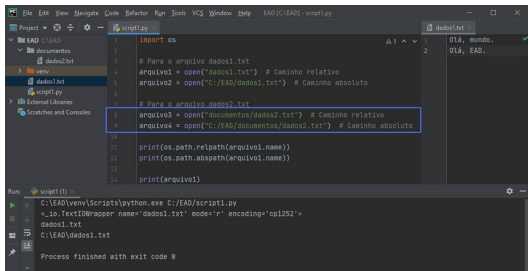
Abertura do arquivo "dados1.txt" pelo caminho relativo.

Utilizamos, na **linha 4**, a função `open` para abrir o arquivo **"dados1.txt"**, que se encontra no mesmo diretório do nosso script. Nessa linha, utilizamos o caminho relativo. Observe que, como o arquivo dados1.txt está na mesma pasta EAD que o script1.py, basta escrever o nome do arquivo como argumento.



Abertura do arquivo "dados1.txt" pelo caminho absoluto.

Abrimos, na **linha 5**, o mesmo arquivo dados1.txt, utilizando o caminho absoluto (completo), que, no nosso exemplo, é: "C:/EAD/dados1.txt".



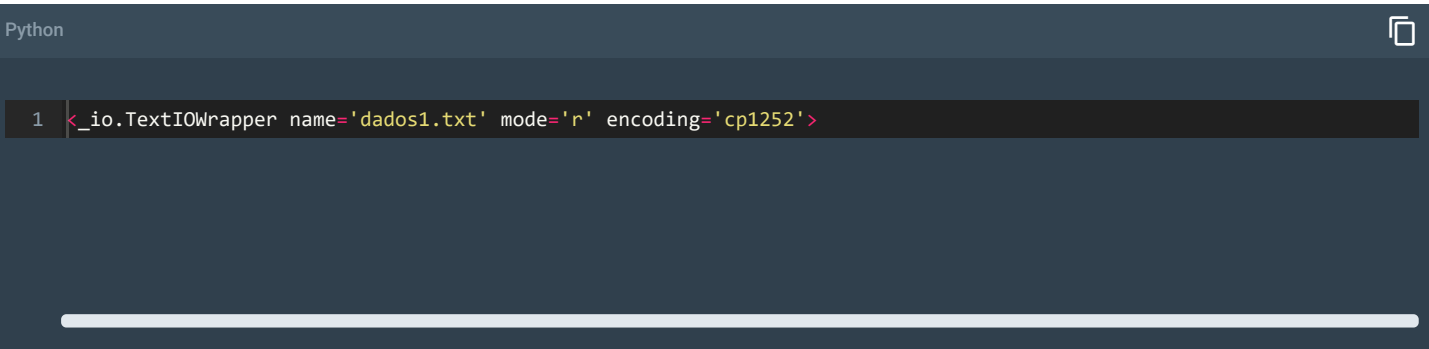
Abertura do arquivo dados2.txt usando os dois caminhos.

Abrimos, **nas linhas 8 e 9**, o arquivo dados2.txt, que se encontra na pasta documentos. Na linha 8, utilizamos o caminho relativo desse arquivo para abri-lo: "documentos/dados2.txt", enquanto, na linha 9, utilizamos o caminho absoluto: "C:/EAD/documentos/dados2.txt".

O Python também disponibiliza algumas funções para exibir os caminhos absolutos e relativos de um arquivo ou diretório, que são:

- Na linha 11, utilizamos a função **path.relpath** para imprimir o caminho relativo do arquivo1, a partir do nome do arquivo passado como parâmetro.
- Na linha 12, utilizamos a função **path.abspath** para exibir o caminho absoluto do mesmo arquivo. Observe que, mesmo utilizando o caminho relativo para abrir o arquivo (linha 4), é possível obter o caminho absoluto utilizando essa função. Isso pode ser verificado na saída do console.
- Na linha 14, utilizamos a função interna **print** para imprimir a variável arquivo1.

Verifique, na saída do console, onde foi impressa a representação do objeto arquivo1:



Desmembrando essa saída, temos:

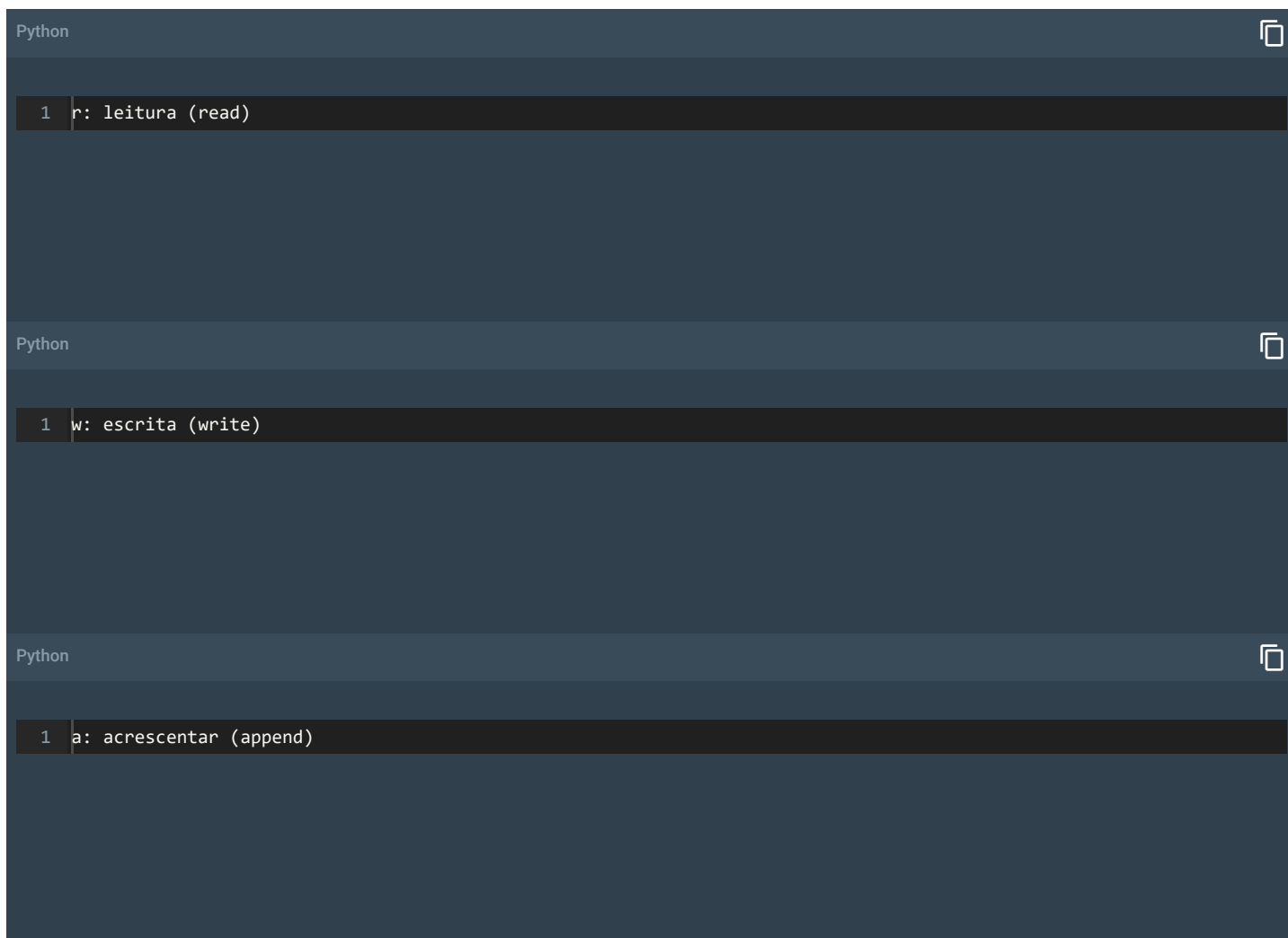
- O tipo do objeto, TextIOWrapper, que trata de arquivos de texto.
- O nome do arquivo, name='dados.txt'.
- O modo de acesso ao arquivo, mode='r'.
- A codificação do arquivo, encoding='cp1252'.

Neste módulo, vamos tratar apenas de arquivos do tipo texto, ou seja, objetos TextIOWrapper. A seguir, vamos apresentar os diferentes modos de acesso aos arquivos.

Modos de acesso a um arquivo

Quando abrimos um arquivo, precisamos informar ao Python o que desejamos fazer, ou seja, qual será o modo (*mode*) de acesso ao arquivo. O modo é um dos parâmetros da função *open*, e cada modo é representado por uma string.

Os principais modos são:



O modo padrão da função *open* é o modo leitura (“r”).

Esses modos podem ser combinados e para informar que desejamos ler e escrever em um arquivo, utilizamos a string “r+”, por exemplo.

O Python também nos permite diferenciar arquivos texto de arquivos binários, como uma imagem, por exemplo. Para informar que desejamos abrir um arquivo binário, adicionamos a string “b” ao modo, ficando “rb”, “wb” e “ab”.

A tabela abaixo resume os modos de acesso a arquivos:

Caractere	Significado
'r'	Abre o arquivo para leitura (default).
'w'	Abre o arquivo para escrita, truncando o arquivo primeiro.

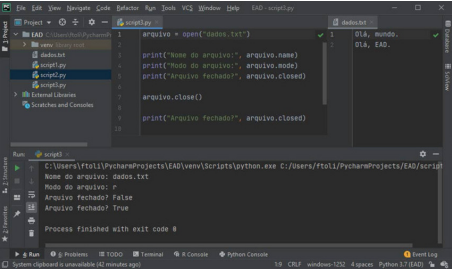
Caractere	Significado
'x'	Cria um arquivo para escrita e falha, caso ele exista.
'a'	Abre o arquivo para escrita, acrescentando conteúdo ao final do arquivo, caso ele exista.
'b'	Modo binário.
't'	Modo texto (default).
'+'	Abre o arquivo para atualização (leitura ou escrita).

Tabela: Modos de abertura de arquivos em Python.
Adaptado de Python (2020).

Atributos do objeto tipo arquivo

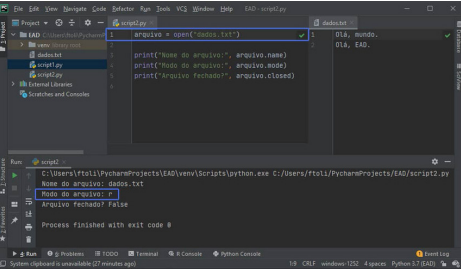
Atributos de um arquivo

O objeto do tipo arquivo contém alguns atributos importantes, como name, mode e closed, veja:



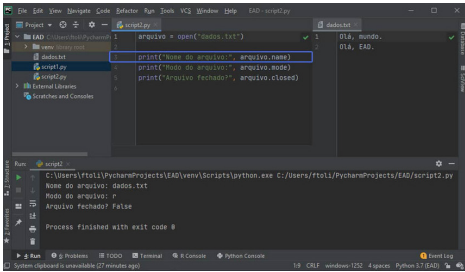
Script2, sua saída e arquivo dados.txt.

Temos o script2, sua saída e arquivo dados.txt.



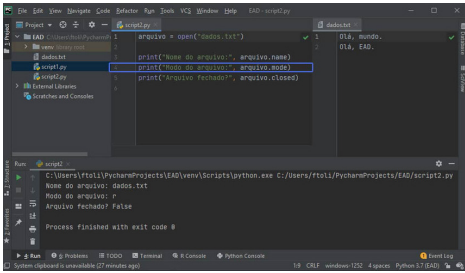
Abertura do arquivo pela função open

Abrimos, **na linha 1**, o arquivo utilizando a função *open*. Como não explicitamos o parâmetro *mode*, o arquivo será aberto no modo leitura ("r").



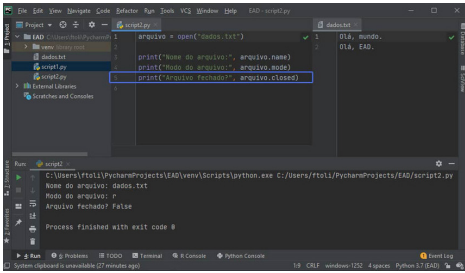
Impressão do atributo *name*.

Imprimimos, **na linha 3**, o atributo *name* do objeto arquivo. Esse atributo contém o nome do arquivo.



Impressão do atributo *name*.

Imprimimos, **na linha 4**, o atributo *mode* do objeto arquivo. Esse atributo contém o modo de acesso do arquivo (r, w, a, rb ...).



Impressão do atributo *closed*.

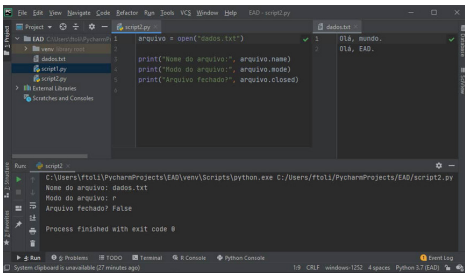
Imprimimos, **na linha 5**, o atributo *closed* do objeto arquivo. Essa atributo serve para verificar se um arquivo está ou não fechado.

Os valores de cada atributo podem ser verificados no console abaixo da imagem.

Fechando um arquivo

Após realizar a operação desejada no arquivo, precisamos liberá-lo. Para isso, utilizamos o método **close()**, que libera a memória alocada pelo interpretador e o uso do arquivo por outros programas, por exemplo.

Agora, vamos utilizar o script do exemplo anterior como base e adicionar uma chamada ao método **close()** e verificar o atributo *closed* novamente:



Script3, sua saída e seu arquivo dados.txt.

Em relação ao script do exemplo anterior, adicionamos, na linha 7, uma chamada ao método `close()` do objeto arquivo.

Na linha 9, imprimimos novamente a propriedade `closed`, onde podemos observar que seu valor agora é `True`.

Lendo o conteúdo de um arquivo

Agora que já sabemos abrir e fechar um arquivo, vamos ver as formas de ler seu conteúdo.

O Python disponibiliza os seguintes métodos para leitura do conteúdo de um arquivo-texto:

Read()

Retorna **todo** o conteúdo de um arquivo como uma única string.

Readline()

Retorna uma **linha** de arquivo, incluindo caracteres de final (`\n` ou `\r\n`), e avança o cursor para a próxima.

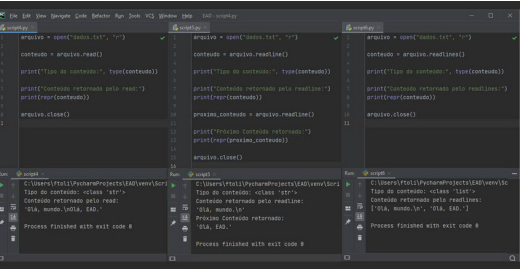
Readlines()

Retorna uma **lista** em que cada item da lista é uma linha do arquivo.

Abaixo, temos **três scripts**, em que cada um utiliza um dos métodos descritos anteriormente para leitura do arquivo. Observe que explicitamos o modo de operação como leitura ("`r`"):

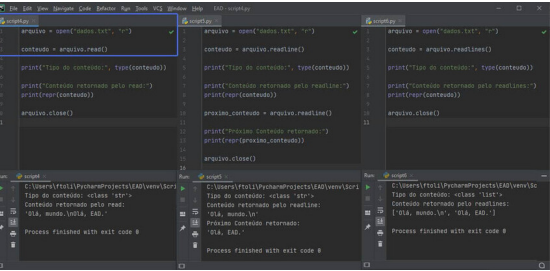
rês scripts

Em cada um dos scripts, vamos abrir o mesmo arquivo `dados.txt` dos exemplos anteriores, ler o conteúdo, verificar o tipo de conteúdo retornado por cada método de leitura e imprimir o valor canônico (real) do conteúdo lido.



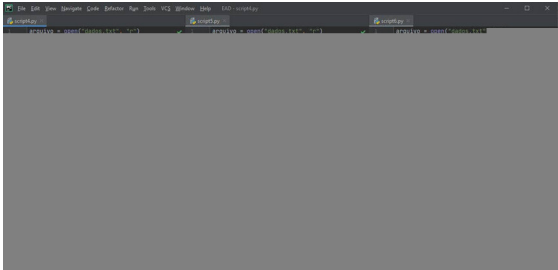
Scripts 4, 5 e 6 e cada uma de suas saídas.

Temos os **scripts 4, 5 e 6** e suas respectivas saídas.



Abertura do arquivo pelo método `read()`.

Abrimos, **no script4.py**, mais à esquerda, abrimos o arquivo na linha 1 e, na linha 3, utilizamos o método `read()` do objeto **arquivo** para ler o conteúdo de `dados.txt` e armazená-lo na variável **conteudo**.



Impressão do atributo *name*.

Verificamos, na linha 5, o tipo do conteúdo retornado pelo método `read()`, utilizando a função interna `type`. Conforme exibido no console, a variável **conteudo** é um objeto do tipo `str` (string).

Impressão do atributo *mode*.

Imprimimos, **na linha 8**, o conteúdo em si, porém utilizamos a função interna **`repr`** para mostrar o conteúdo real contido da variável **conteudo**. Observe que foi retornado todo o texto existente no arquivo `dados.txt`, que também pode ser verificado no console.

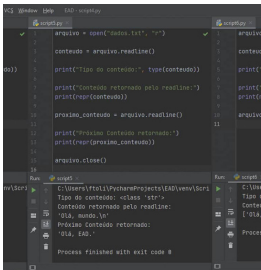
No **script5.py**, seguimos os mesmos passos do script anterior, porém, na linha 3, utilizamos o método `readline()`.

Na linha 5, verificamos que o tipo do conteúdo retornado pelo método `readline()` também é um objeto do tipo `str` (string).

Na linha 8, imprimimos a representação do conteúdo que contém apenas a **primeira linha** do arquivo `dados.txt` (incluindo o caractere de final de linha `\n`). Isso aconteceu porque, quando abrimos o arquivo utilizando o modo leitura (`'r'`), o cursor interno de leitura fica posicionado no início do arquivo.

Se chamarmos novamente o método `readline()`, linha 10, será retornado à próxima linha do arquivo, que foi impressa na linha 13. Confira a saída do script 5 no console abaixo dele; seguimos os mesmos passos do script anterior, porém, na linha 3, utilizamos o método `readline()`.

Veja tudo isso na imagem a seguir:



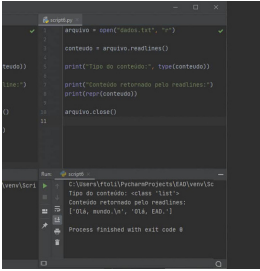
Script5 e sua saída.

No **script6.py**, seguimos novamente os mesmos passos, mas, desta vez, utilizamos o método `readlines()`.

Na linha 5, verificamos que o tipo do conteúdo retornado pelo método `readlines()` é um objeto do tipo `list` (lista).

Na linha 8, imprimimos o conteúdo retornado, que é uma lista na qual **cada item é uma linha do arquivo**. Veja a saída desse script no console abaixo dele.

Além dos três métodos já apresentados, os objetos do tipo arquivo são iteráveis. Com isso, podemos utilizar o laço **for** diretamente sobre os objetos desse tipo. Veja tudo isso na imagem a seguir:

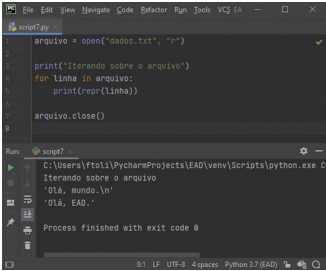


Script6 e sua saída.

Veja agora como iterar diretamente sobre um arquivo:

Na linha 1, abrimos o arquivo da mesma maneira que fizemos nos exemplos anteriores.

Na linha 4, utilizamos o laço for para iterar diretamente sobre a variável arquivo. Para cada iteração, recebemos uma nova linha do arquivo, disponibilizada na variável linha, impressa na linha 5. Observe a saída do console abaixo da imagem:



Script7 e sua saída.

Quando precisamos abrir um arquivo muito grande, é inviável utilizar os métodos *read* e *readlines*, pois eles retornam todo o conteúdo do arquivo de uma só vez, seja na forma de string, seja na forma de lista. Isso pode consumir todos os recursos do computador, travando seu programa.

Nesses casos, precisamos chamar o método *readline* inúmeras vezes até o final do arquivo ou iterar diretamente sobre o objeto do tipo arquivo.

Após utilizar qualquer um dos métodos para leitura do arquivo apresentado, não podemos utilizá-los novamente. Isso acontece porque o cursor estará posicionado ao final do arquivo, e as chamadas aos métodos *read*, *readline* ou *readlines* retornarão vazias.



Atividade discursiva

Para situações em que precisamos ler o conteúdo de um arquivo mais de uma vez, temos duas opções. Na sua opinião, quais são elas?

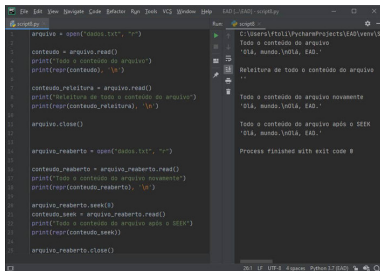
Digite sua resposta aqui...

Exibir solução ▾

Fechar e abrir novamente o arquivo.

Utilizar o método `seek(n)`, passando como argumento o número da linha onde desejamos posicionar o cursor. A chamada `seek(0)` retorna o cursor para o início do arquivo.

Depois de conhecer a resposta correta, confira o exemplo do script8, onde exploramos na prática essa situação:



```
1 arquivo = open('dados.txt', 'r')
2
3 conteudo = arquivo.read()
4 print(f"Imo o conteúdo do arquivo")
5 print(repr(conteudo), "\n")
6
7 conteudo_releitura = arquivo.read()
8 print(f"Releitura de todo o conteúdo do arquivo")
9 print(repr(conteudo_releitura), "\n")
10
11 arquivo.close()
12
13 arquivo_reaberto = open('dados.txt', 'r')
14
15 conteudo_reaberto = arquivo_reaberto.read()
16 print(f"Imo o conteúdo do arquivo novamente")
17 print(repr(conteudo_reaberto), "\n")
18
19 arquivo_reaberto.seek(0)
20 conteudo_seek = arquivo_reaberto.read()
21 print(f"Imo o conteúdo do arquivo mais o SEEK")
22 print(repr(conteudo_seek), "\n")
23
24 arquivo_reaberto.close()
```

Output:

```
Imo o conteúdo do arquivo
'dia, mundo, \n', b'\n'

Releitura de todo o conteúdo do arquivo
'dia, mundo, \n', b'\n'

Imo o conteúdo do arquivo novamente
'dia, mundo, \n', b'\n'

Imo o conteúdo do arquivo mais o SEEK
'dia, mundo, \n', b'\n'

Process finished with exit code 0
```

Script8 e sua saída.

Vejamos agora o que se sucedeu em cada linha a seguir:

Na linha 1

Abrimos o arquivo em modo leitura.

Na linha 3

Lemos todo seu conteúdo utilizando o método `read`, que é impresso na linha 5. Veja no console à direita da imagem.

Na linha 7

Utilizamos o método `read` para ler novamente o conteúdo do arquivo e atribuímos o valor retornado à variável `conteudo_releitura`. Na linha 9, imprimimos o conteúdo dessa variável, que, conforme exibido no console, é uma string vazia (`""`).

Para contornar esse problema, fechamos e abrimos o arquivo novamente, linhas 11 e 14, respectivamente. Na linha 16, utilizamos novamente o método `read` e imprimimos o conteúdo retornado na linha 18. Observe que, mais uma vez, conseguimos acessar todo o conteúdo do arquivo.

Para demonstrar a utilização do método `seek`, no mesmo arquivo que já estava aberto, `arquivo_reaberto`, utilizamos o método `seek(0)`, linha 20. Imprimimos mais uma vez o conteúdo correto do arquivo na linha 23.

Toda a sequência pode ser acompanhada pelo console.

Atenção!

Todos os três métodos apresentados aceitam como parâmetro a quantidade de *bytes* que desejamos ler.

O valor padrão para esse parâmetro é -1, o que corresponde a todo o arquivo.

Escrevendo conteúdo em um arquivo

Confira agora como escrever conteúdo em um arquivo a partir da função `open`. Vamos lá!

A primeira modificação é alterar o modo de acesso ao arquivo. Para escrita de texto, podemos utilizar o modo `w` (*write*) ou o modo `a` (*append*), a seguir:

- O modo **w** abre o arquivo para escrita, truncando o arquivo em primeiro lugar. Caso ele não exista, será criado um.
- O modo **a** abre o arquivo para escrita, acrescentando conteúdo ao final dele, caso ele exista; do contrário, será criado um arquivo.

O Python disponibiliza dois métodos para escrita de conteúdo em um arquivo texto, para o modo **w** e para o modo **a**. Os métodos *write* e *writelines* são descritos abaixo:

Write (texto)

Escreve todo o conteúdo passado como parâmetro no arquivo.

Writelines (iterável)

Escreve cada item do iterável (exemplo: lista) no arquivo.

No exemplo a seguir, vamos criar dois scripts para mostrar o uso do modo `w`. No primeiro, `script9`, vamos utilizar o método *write*. No segundo, `script10`, vamos utilizar o método *writelines*.

Scripts 9 e 10 e suas saídas.

Temos os scripts 9 e 10 e suas respectivas saídas.

Abertura do arquivo `dados_write.txt`.

Abrimos, **no script9**, o arquivo `dados_write.txt` para escrita utilizando o modo `w` na linha 1.

Escrita dos conteúdos pelo método *write*.

Escrevemos os conteúdos utilizando o método *write*, conforme linhas 2 e 3, e fechamos o arquivo, como exposto na linha 4.

Resultado do arquivo dados_write.txt.

Observe como ficou o arquivo dados_write.txt, abaixo do script9, após a execução desse script.

Resultado do arquivo dados_write.txt.

Criamos, **no script10**, uma lista chamada **linhas** na linha 1. Abrimos o arquivo dados_write.txt para escrita na linha 4, utilizando o mesmo modo de acesso ao arquivo, modo w. Para escrever o conteúdo da lista **linhas** no arquivo, utilizamos o método writelines, linha 5.

Resultado do conteúdo do arquivo dados_writelines.txt.

Verifique também o conteúdo do arquivo dados_writelines.txt após a execução do script, abaixo do script10 na imagem.

Fique atento às seguintes orientações:

- O Python não insere quebra de linha ('\n') entre os elementos da lista. Precisamos fazer isso **manualmente!**
- Como o modo w trunca o arquivo, ou seja, remove todo o conteúdo do arquivo, caso ele exista, podemos executar esses scripts repetidas vezes e, ainda assim, o resultado será sempre o mesmo.

No próximo exemplo, vamos mostrar como utilizar o modo append (a) para adicionar conteúdo a um arquivo já existente.

Para isso, vamos abrir o arquivo dados_write.txt, criado pelo script9, utilizando o modo a. Utilizamos esse modo para acrescentar conteúdo a um arquivo. Confira o próximo script:

Script 11 e saída.

Temos o script11 e sua saída.

Abertura do arquivo dados_write.txt pelo append (a).

Abrimos, em primeiro lugar, na linha 1, o arquivo dados_write.txt utilizando o modo escrita a (append).

Utilização do método *write*.

Utilizamos, **na linha 3**, o método *write* para acrescentar o texto "\nConteúdo adicional." ao final do arquivo dados_write.txt.

Fechamento do arquivo.

Fechamos, **na linha 5**, o arquivo.

Resultado do conteúdo final do arquivo.

Observe como ficou o conteúdo final do arquivo à direita da imagem, onde a nova frase foi posicionada corretamente ao final do arquivo.

Boas práticas

Ao lidar com arquivos, devemos utilizar a palavra reservada *with*, disponibilizada pelo Python. Ela garante que o arquivo será fechado adequadamente após utilizarmos o arquivo, não sendo necessário chamar o método `close` explicitamente. A sintaxe de utilização do *with* é:

```
Python
```

```
1 with open(caminho, modo) as nome: (seu código indentado)
```

Iniciamos com a palavra reservada *with*, seguida da função *open*, a palavra reservada *as*, um nome de variável que receberá o objeto do tipo arquivo e dois pontos. Todo o código indentado posteriormente está dentro do contexto do *with*, no qual o arquivo referenciado pela variável *nome* estará disponível.

Veja como utilizar o *with* no exemplo a seguir. Clique nas setas e acompanhe:

Script 12 e saída.

Temos o script12 e sua saída.

Abertura do arquivo dados.txt no modo leitura.

Utilizamos a sintaxe do *with*, na linha 3, e o arquivo dados.txt é aberto no modo leitura, atribuído à variável arquivo. Esta variável está disponível em todo o escopo do *with*, linhas 4, 5 e 6.

Iteração sobre o conteúdo do arquivo.

Iteramos, **nas linhas 4 e 5**, sobre o conteúdo do arquivo e imprimimos linha por linha.

Impressão do nome do arquivo.

Imprimimos, **na linha 6**, o nome do arquivo.

Saídas no console à direita.

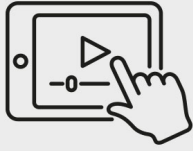
Verifique as saídas no console à direita.



Manipulação de arquivos no Python

Veja a seguir como acontece a manipulação de arquivos no Python. Mas antes, para acompanhar o vídeo e poder executar o programa, baixe os arquivos neste [link](#).

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Na imagem a seguir temos o script `exercicio_1_modulo_1.py` e o arquivo de texto `exercicio_1.txt`.

```
exercicio_1_modulo_1.py
1 arquivo = open("exercicio_1.txt", "w")
2 arquivo.write("\nTeste 3")
3 arquivo.write("\nTeste 4")
4 arquivo.close()
5
6 arquivo = open("exercicio_1.txt", "a")
7 arquivo.write("\nTeste 5")
8 arquivo.write("\nTeste 6")
9 arquivo.close()
10
11 arquivo = open("exercicio_1.txt", "w")
12 arquivo.write("\nTeste 7")
13 arquivo.write("\nTeste 8")
14 arquivo.close()

exercicio_1.txt
1 Teste 1
2 Teste 2
3
```

Script `exercicio_1_modulo_1.py` e arquivo de texto `exercicio_1.txt`.

Identifique, nas alternativas a seguir, qual o conteúdo do arquivo `exercicio_1.txt` após executar o script. Desconsidere eventuais linhas em branco:

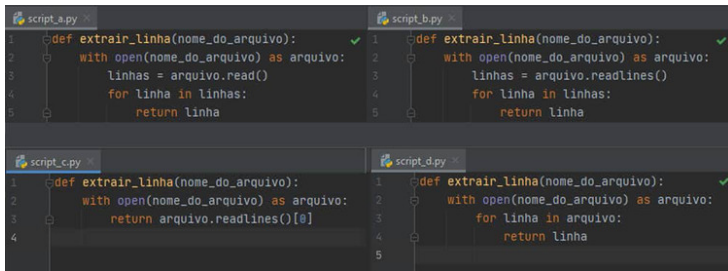
- A Teste5 - Teste6 - Teste7 - Teste8
- B Teste7 - Teste8
- C Teste1 - Teste2 - Teste7 - Teste8
- D Teste1 - Teste2 - Teste3 - Teste4 - Teste5 - Teste6 - Teste7 - Teste8
- E Teste3 - Teste4 - Teste5 - Teste6 - Teste7 - Teste8

Parabéns! A alternativa B está correta.

Sempre que utilizamos o modo de acesso `w` para abrir um arquivo, o conteúdo do arquivo é apagado (truncado). Na linha 11, realizamos a abertura do arquivo utilizando esse modo, excluindo todo conteúdo existente. Nas linhas 12 e 13, escrevemos o conteúdo final que constará no arquivo: Teste 7 e Teste 8.

Questão 2

Imagine que fomos contratados para desenvolver um programa em que uma das funcionalidades é extrair a primeira linha de um arquivo. Durante os testes, verificamos que a função que desenvolvemos para extrair as linhas, `extrair_linha`, precisava ser refeita, pois, como os arquivos eram muito grandes, a função consumia muita memória, travando o programa.



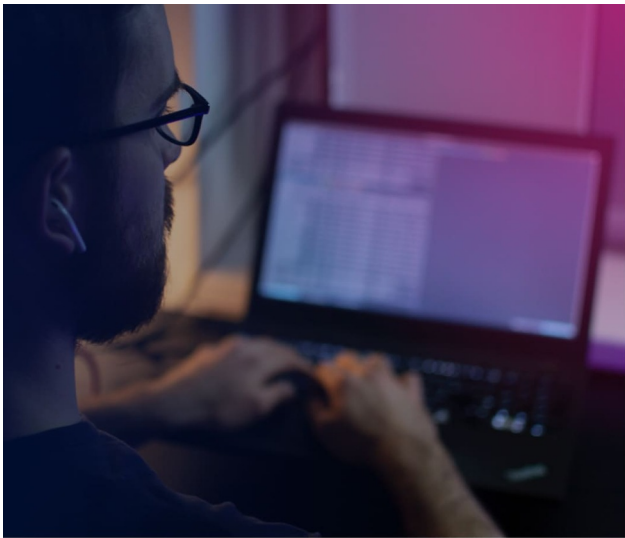
Desenvolvimento do programa.

Dentre as opções de scripts apontadas na imagem, identifique a alternativa que atende ao problema apresentado:

- A script_a.py
- B script_b.py
- C script_c.py
- D script_d.py
- E script_a.py e script_b.py

Parabéns! A alternativa D está correta.

Quando lidamos com arquivos muito grandes, devemos utilizar o método `readline()` ou iterar diretamente sobre o arquivo. Esses métodos retornam uma linha por vez do arquivo, sem ler todo seu conteúdo, resolvendo o problema de leitura de arquivos muito grandes.

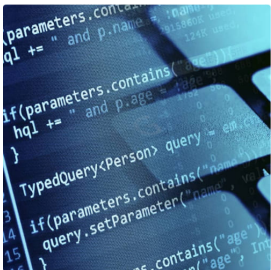


2 - Funções de manipulação de strings

Ao final deste módulo, você será capaz de reconhecer as funções de manipulação de strings.

Manipulação de strings

Métodos de manipulação de strings

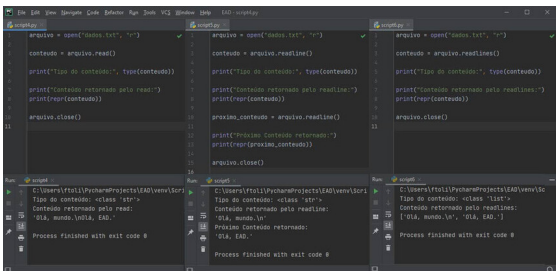


Durante a vida de programador, é muito comum nos depararmos com situações em que precisamos realizar alguns ajustes e operações sobre os textos lidos de arquivos, como remover espaço em branco, colocar todas as letras maiúsculas, substituir e contar palavras.

Neste módulo, veremos alguns métodos presentes nos objetos do tipo str (string), que são muito utilizados em conjunto com a manipulação de arquivos.

Método *strip*

Como mostrado nos scripts da imagem abaixo (script5), ao ler o conteúdo do arquivo, o Python retorna os caracteres de final de linha (`\r` e `\n`). Muitas vezes, essa informação não é necessária, principalmente se estivermos tratando uma linha de cada vez. Veja:



Dependendo do objetivo, esses caracteres são considerados lixo e podem atrapalhar o processamento que desejamos realizar. Para remover esses caracteres e também espaços em branco adicionais, o tipo `str` disponibiliza o método `strip()`. O método *strip* remove os caracteres do início e do final da linha. Observe o uso deste método no exemplo a seguir. Clique nas setas para acompanhar cada ação:

Scripts 13a e 13b, suas saídas e arquivo dados13.txt.

Temos os **scripts 13a e 13b**, suas respectivas saídas e seu arquivo dados13.txt.

Abertura do arquivo dados13.txt.

Abrimos o arquivo **dados13.txt**, exibido à direita da figura, e imprimimos o conteúdo de cada linha desse arquivo. Vamos utilizar o método `repr` para verificar o real conteúdo da string.

Separação do script em duas partes.

Separamos o script em duas partes, **a** e **b**, de forma a se perceber melhor o resultado da utilização do método `strip()`.

Abertura do arquivo em modo leitura.

Abrimos o arquivo em modo leitura, na primeira parte, **script13a**, utilizando o `with` na linha 1, iteramos sobre o objeto **arquivo** na linha 3 e imprimimos o conteúdo de cada linha na linha 4.

Presença dos espaços em branco e caracteres de nova linha.

Observe, no console abaixo do **script13a**, que os espaços em branco no início de cada linha e os caracteres de nova linha (`\n`) estão presentes na string.

Abertura e iteração do mesmo arquivo.

Abrimos, na segunda parte, **script13b**, o mesmo arquivo e iteramos da mesma forma. Porém, desta vez, “limpamos” a linha utilizando o método `strip`, na linha 4. Esse método retorna uma nova string, que é armazenada na variável `linha_limpa`. Em seguida, na linha 5, imprimimos a representação dessa variável.

Saída do console.

Observe a saída do console e verifique que os caracteres em branco do início da linha e os caracteres de nova linha foram removidos.

Um exemplo real de utilização do `strip` é quando desejamos contar a quantidade de linhas com algum conteúdo em um arquivo. Dependendo do tamanho do arquivo, é inviável remover manualmente as linhas em branco. Para resolver esses problemas, podemos utilizar o método `strip` da seguinte forma:

Vamos abrir o mesmo arquivo do exemplo anterior, `dados13.txt`, e iterar sobre cada linha, incrementando um contador, que será impresso no final. Para melhor ilustrar, vamos mostrar como contar as linhas sem usar o método `strip`, `script14a`, com o `strip` `script14b`, veja:

Scripts 14a e 14b e suas respectivas saídas.

Temos os scripts 14a e 14b e suas respectivas saídas.

Abertura do arquivo em modo leitura.

Abrimos, no primeiro script, o arquivo em modo leitura na linha 1, criamos e inicializamos uma variável inteira **contador**, linha 3, e iteramos seu conteúdo linha a linha, utilizando o loop for, linha 4.

Verificação da variável conter ou não conteúdo.

Verificamos, na linha 5, se a variável linha, do tipo str, contém algum conteúdo. Lembrando que uma variável do tipo string testa para falso apenas se seu conteúdo for vazio (" ou ""); caso a string contenha, pelo menos, um espaço, ela testará verdadeiro. Continuando, na linha 6, caso exista algum conteúdo, incrementamos o contador.

Obtenção do valor 7 pelo contador.

O contador, para o primeiro script, obteve valor 7, indicando que o arquivo contém 7 linhas, conforme console abaixo do script14a.

Escrita do mesmo código.

Escrevemos, no segundo script, praticamente o mesmo código, iniciando pela abertura do arquivo na linha 1, e assim por diante. A exceção é a linha 5, que testa o conteúdo da variável **linha** após utilização do método **strip()**. Observe que, agora, contamos corretamente o número de linhas com algum conteúdo, 3.

Métodos *count* e *split*

Outra atividade muito comum na manipulação de arquivos é a contagem do número de vezes que determinada palavra aparece. O Python disponibiliza o método *count* para strings, que recebe como parâmetro a palavra que desejamos contar e retorna o total de ocorrências dela. Sua sintaxe é a seguinte:

```
Python
```

```
1 contagem = variavel_string.count(palavra)
```

Nela, a `variavel_string` é uma variável do tipo `str` e `palavra` é a string que desejamos contar.

Veja a seguir como utilizamos o método `count` para contar a quantidade de vezes em que aparece a palavra “Olá”, no arquivo `dados13.txt`:

Scripts 15, saída e arquivo `dados13.txt`.

Temos os scripts 15, sua saída e arquivo `dados13.txt`.

Aplicação do método *read*.

Utilizamos, após abrir o arquivo na linha 1, o método *read*, linha 2, para retornar todo o conteúdo do arquivo como uma única string e armazená-lo na variável *texto*. Na linha 3 utilizamos o método *count* da variável *texto*, passando como argumento à string “Olá”.

Armazenamento do total de ocorrências de “Olá”.

O total de ocorrências da palavra “Olá” foi armazenado na variável **contador**, que foi impressa na linha 4. Observe que o total de ocorrência está correto: 3.

Apesar de ser muito simples a utilização do método *count* do tipo *str*, pode gerar alguns efeitos indesejáveis, pois esse método também conta as palavras que contêm parte da string passada como argumento.

Exemplo

Considere a frase: “Eu amo comer amoras no café da manhã”. Se utilizarmos o método *count*, com a string “amo” como argumento, o retorno será 2, pois o método irá considerar tanto a palavra *amo* quanto a palavra *amoras*.

Para contornar esse problema, podemos “quebrar” uma frase em palavras e depois verificar se cada palavra é igual à string que buscamos.

Isso nos leva a outro método muito utilizado em processamento de textos, o método *split()*, que é usado para quebrar uma string em partes menores, retornando uma lista com essas partes. Sua sintaxe é a seguinte:

Python



```
1 lista_termos = variavel_string.split(separador)
```

No exemplo, a *variavel_string* é uma variável do tipo *str*, e *separador* é uma string que desejamos utilizar como ponto de quebra do texto. O retorno desse método é uma lista de strings.

Digamos que desejamos usar o método *split* com separador ‘-’ na frase: “Amo futebol – Gosto de basquete”. O resultado seria uma lista em que o primeiro elemento é a string “Amo futebol ” e o segundo elemento é string “ Gosto de basquete”. Caso nenhum separador seja passado como argumento, o Python irá utilizar um ou mais espaços como separador padrão.

Dica

A string utilizada como separador **não** aparecerá em nenhum elemento da lista retornada.

No exemplo a seguir o método *split* é utilizado em três frases diferentes para mostrar o comportamento dele. Confira:

Script 16 e saída.

Temos o script 16 e sua saída.

Aplicação do método *split*.

Utilizamos o método *split* sem argumentos, linha 2, e imprimimos a lista retornada na linha 3. Observe pelo console que cada item da lista é uma palavra da frase.

Aplicação do método *split* e impressão da lista.

Utilizamos o método *split*, também, sem argumentos, linha 6, e imprimimos a lista retornada na linha 7. Observe pelo console que, mesmo havendo muitos espaços em branco entre as palavras da frase, a lista contém apenas as palavras, sem nenhum espaço adicional. O Python é esperto o suficiente para detectar vários espaços contínuos e tratá-los como um único separador.

Aplicação do método *split* passando uma vírgula (,).

Temos, na linha 9, a última frase: "Carro,moto,avião". Desta vez, utilizamos o método *split* passando uma vírgula (,) como argumento, na linha 10. O resultado é uma lista contendo apenas as palavras, **sem o separador**, conforme podemos ver pelo console.

Agora que sabemos como funciona o método *split*, no próximo exemplo, vamos utilizar esse método para realizar a contagem da palavra "amo" na frase do exemplo anterior: "Eu amo comer amoras no café da manhã.". Vamos aproveitar para comparar os resultados obtidos pelos métodos *count*

e *split*.

Script 17 e saída.

Temos o script 17 e sua saída.

Definição da variável frase.

Definimos, na linha 1, nossa variável **frase** com o conteúdo descrito anteriormente.

Aplicação do método *count*.

Utilizamos, na linha 4, o método *count* do tipo str para contar diretamente o número de ocorrências da string “amo” na frase. Pelo console, vemos que o resultado da contagem foi 2. Uma ocorrência pela palavra amo e outra pela palavra **amora**.

Criação de um contador e aplicação do método *split*.

Criamos, para realizar a contagem da mesma frase usando o *split*, um contador, linha 7, e quebramos a frase em palavras utilizando o método *split*, linha 8.

Iteração sobre cada palavra.

Iteramos, na linha 9, sobre cada palavra utilizando a variável **termo**. Para cada termo, comparamos se seu valor é igual a string “amo”, linha 10. Caso seja igual, incrementamos o contador, linha 11. Observe que desta vez alcançamos o resultado correto para a contagem, 1.

No script17.py, para realizar a contagem da palavra “amo” na frase “Eu amo comer amoras no café da manhã”, foi criada uma lista chamada lista_termos, através do uso do método *split* sobre a variável que continha a frase mencionada. Também foi criado um contador e houve uma iteração sobre lista_termos, realizando comparações entre os elementos desta e a palavra “amo”.

Será que não haveria uma maneira de realizar essa contagem de outra maneira, fazendo uso dos métodos de manipulação de strings apresentados até aqui?

Utilize o emulador de códigos abaixo para desenvolver outra solução para o problema da contagem da palavra “amo” na frase “Eu amo comer amoras no café da manhã.”:

Exercício 1

TUTORIAL COPIAR

Python3

```
1 frase = "Eu amo comer amoras no café da manhã."
2 lista_termos = frase.split()
3
```

null

null



A resolução está contida a seguir:

Exercício 2

TUTORIAL COPIAR

Python3

```
1 frase = "Eu amo comer amoras no café da manhã."
2 lista_termos = frase.split()
3 contagem = lista_termos.count("amo")
4 print("Contagem = ", contagem)
```

null

null



No código acima, na linha 2, temos a separação das palavras da frase por meio do uso do método *split* na variável *frase* (tipo *string*). A seguir, na linha 3, utilizamos o método *count* com o argumento “amo”, para obtermos a quantidade de ocorrências dessa *string* (“amo”) em *lista_termos*. Por fim, na linha 4, imprimimos o resultado no console do emulador.

Vamos avançar para mais um método!

Método *join*

Assim como há a necessidade de quebrar uma frase em uma lista de palavras, podem existir situações em que ocorra o inverso, ou seja, temos uma lista de palavras ou frases e desejamos concatená-las em uma única *string*.

Para essa operação, utilizamos o método *join* do tipo *str*, que retorna uma única *string* com todos os elementos da lista concatenados, utilizando determinado conector. Sua sintaxe é a seguinte:

Python



```
1 string_final = “conector”.join(iteravel)
```

Em que “conector” é a *string* que será utilizada entre os elementos da lista que serão concatenados (ex. ‘, ’) e *iteravel* é um iterável, como uma lista ou tupla.

No exemplo a seguir, vamos unir os elementos de uma lista utilizando dois conectores diferentes: o conector vírgula (‘, ’) e o conector de nova linha (‘\n’). Após a união, vamos gravar o conteúdo em um arquivo para mostrar o resultado:

Script 19, arquivos *texto1.txt* e *texto2.txt*.

Temos o script 19, arquivos *texto1.txt* e *texto2.txt*.

Criação da lista `minha_lista`.

Criamos, na linha 1, a lista `minha_lista`, que será utilizada nos dois exemplos.

Aplicação do método `join`.

Utilizamos, na linha 3, o método `join` com o conector `,` e atribuímos o resultado à variável `texto1`. O resultado dessa variável foi gravado no arquivo `texto1.txt`, que pode ser visto na imagem.

Junção dos elementos da mesma lista.

Fazemos, na linha 7, a junção dos elementos da mesma lista utilizando o conector `\n`. O resultado da junção foi gravado no arquivo `texto2.txt`, que pode ser visto à direita da imagem. Com isso, fomos capazes de colocar cada elemento da lista em uma linha do arquivo.

Formatação de strings

Manipulação de variáveis em strings

Até o momento, realizamos operações sobre strings pré-existentes. No entanto, é muito comum precisarmos juntar valores de variáveis com strings.

Agora, veremos algumas funções relacionadas às strings, começando pela formatação de strings (*string formatting*).

A formatação de strings permite ajustar como uma string será exibida ou gravada em um arquivo, por exemplo. Ajustes como: número de casas decimais em *float*, exibição de datas, inclusão de variáveis e espaçamento são alguns dos exemplos.

Existem basicamente três formas de realizar a formatação de strings. São elas:

- Utilizando **f-strings** (*formatted string literals*).
- Utilizando o método `format()` das strings.
- Fazendo manualmente.

f-strings

São tipos especiais de strings que aceitam uma notação especial para permitir a inclusão de expressões diretamente na string.

Veremos como utilizar f-strings, recurso adicionado na versão 3.6 do Python.

F-strings

O objetivo principal da criação das f-strings foi facilitar a formatação de strings.

Para definimos uma variável f-string, precisamos incluir, antes das aspas que definem uma string, a letra `f` (ou `F`), por exemplo:

```
Python
```

```
1 minha_string = f"Olá Mundo {expr}"
```

Dentro das f-string, podemos utilizar expressões em Python entre as chaves, delimitadas pelos caracteres `{}` e `}`.

Essas expressões incluem variáveis ou **literals**. Inclusive, podemos fazer chamada para funções ou utilizar métodos de variáveis dentro desses delimitadores.

Todo o conteúdo entre os caracteres `{}` e `}` é substituído pelo resultado da expressão e interpolado à string.

Compare a seguir a manipulação manual de strings com a utilização de f-string e veja alguns exemplos de uso de literais mais sofisticados dentro de f-strings:

literals

É algo que o interpretador reconhece como uma sintaxe válida (exemplo: `2 > 3` retorna `False`, `[1, 2]` retorna uma lista).

Script 20 e saída.

Temos o script 20 e sua saída.

Definição da variável nome.

Definimos, na linha 1, a variável **nome**, do tipo string, que será utilizada ao longo do script.

Mostramos, na linha 3, como incluir o valor da variável **nome** no meio de outra string, utilizando o processo manual de concatenação de strings por meio do operador '+'.

Aplicação da sintaxe f-string.

Utilizamos, na linha 4, a sintaxe da f-string para incluir o valor da variável `nome` também no meio de outra string. Observe que a IDE PyCharm já detectou que temos uma expressão entre as chaves e alterou a cor para destacar esse elemento. No console, o resultado das variáveis `minha_string` e `minha_fstring1` foi o mesmo, porém a sintaxe da f-string é muito mais clara e simples de ser utilizada e entendida.

Aplicação do método *upper*.

Temos, na linha 5, o mesmo exemplo da linha anterior, porém chamamos o método *upper* do tipo *str* para colocar todas as letras em maiúsculo. Isso foi feito diretamente na expressão!

Aplicação de soma na expressão entre chaves.

Utilizamos, na linha 6, uma soma na expressão entre chaves, que foi calculada corretamente, conforme console.

Aplicação do comparador booleano >.

Utilizamos, na linha 7, o comparador booleano >, que também foi avaliado corretamente.

Aplicação do operador IN.

Utilizamos, na linha 8, o operador IN para verificar a pertinência de um número em uma lista, que, mais uma vez, foi avaliado corretamente. Observe como é fácil e intuitiva a utilização de f-string!

Veja agora algumas funcionalidades adicionais de formatação de string utilizando f-string, como a definição de largura de uma string, formatação de float e de datas:

```
1 from datetime import datetime
2
3 frutas = ['abacaxi', 'laranja', 'ma', 'banana']
4 for fruta in frutas:
5     minha_fruta = f'Nome: {fruta[0]} - Número de letras: {len(fruta): 1}'
6     print(minha_fruta)
7
8 print()
9
10 pi = 3.1415
11 meu_numero = f'O número PI é: {pi:.1f}'
12 meu_numero_destacado = f'O número PI destacado é: {pi:6.1f}'
13 meu_numero_preciso = f'O número PI mais preciso é: {pi:.4f}'
14 print(meu_numero)
15 print(meu_numero_destacado)
16 print(meu_numero_preciso)
17
18 print()
19
20 data = datetime.now()
21 minha_data = f'A data de hoje é {data}'
22 minha_data_formatada = f'A data de hoje formatada é {data:%d/%m/%Y}'
23 print(minha_data)
24 print(minha_data_formatada)
```

```
Nome: abacaxi - Número de letras: 10
Nome: laranja - Número de letras: 7
Nome: ma - Número de letras: 3
Nome: banana - Número de letras: 6

O número PI é: 3.1
O número PI destacado é: 3.1
O número PI mais preciso é: 3.1415

A data de hoje é 2020-08-13 18:50:32.262037
A data de hoje formatada é 13/08/2020

Process finished with exit code 0
```

Script 21 e sua saída.

Para facilitar o entendimento, o script21 foi dividido em três trechos. O primeiro trecho, entre as linhas 3 e 6, trata da formatação de largura do conteúdo de expressões, que serve, principalmente, para alinhar conteúdo de texto. No segundo trecho, das linhas 10 a 16, mostramos como formatar floats. No terceiro trecho, das linhas 20 a 24, mostramos como formatar datas. Confira cada trecho citado:

Linhas 3 até 6

No primeiro trecho, definimos uma lista chamada **frutas** na linha 3 e, na linha 4, percorremos cada item dessa lista.

Para cada item, montamos a f-string `minha_fruta`, que contém o nome da fruta e o número de letras que a fruta tem. Destacamos essa linha a seguir:

```
minha_fruta = f"Nome: {fruta:12} - Número de letras: {le
```

Variável Número de espaços Va

Destaque f-string.

Para indicar a largura, ou melhor, o número de espaços que o conteúdo de uma variável deve ocupar, devemos utilizar a sintaxe {variavel:n}, onde temos o nome da variável, seguida de dois pontos (:) e o número de espaços (n) que se deve ocupar.

Caso o tamanho do conteúdo da variável seja menor que o número n, serão incluídos espaços em branco até completar esse tamanho. A posição dos espaços adicionados depende do tipo da variável. Para variáveis do tipo string, os espaços são adicionados à direita, enquanto para variáveis do tipo inteiro, os espaços são adicionados à esquerda.

Caso o tamanho do conteúdo da variável seja maior que o número n, a formatação será ignorada.

Retornando ao exemplo, desejamos imprimir o nome da fruta de forma que ela ocupe 12 espaços ({fruta:12}), e o número de letras da fruta deve ocupar apenas três espaços ({len(fruta):3}). Observe o resultado obtido no console à direita.

Linhas 10 até 16

No segundo trecho, definimos uma variável *float* na linha 10 e criamos três f-strings para exibir esse conteúdo.

A formatação com f-string nos permite um controle maior de como será exibido um número do tipo *float*, no qual podemos definir a largura e o número de casas decimais que devem ser exibidos. A sintaxe para formatar um *float* é a seguinte:

Python



```
1 {variavel_float:largura.precisao f}
```

Pelo exemplo, temos o nome da variável do tipo *float* seguida de dois pontos (:), a largura total que o número deve ocupar, incluindo as casas decimais, e o ponto (separador de decimal), seguido de um ponto (.), o número de casas decimais (precisao) e a letra "f", que deve estar junto à precisão. A largura é opcional.

Na primeira f-string, na linha 11, utilizamos a expressão {pi:.1f}, ou seja, queremos que seja exibido o valor da variável *pi* com uma casa decimal apenas. Como não especificamos a largura, ela será calculada de forma a acomodar toda a parte inteira do *float*.

Na f-string da linha 12, utilizamos a expressão {pi:6.1f}, que indica que o número deve ocupar seis espaços, sendo que, necessariamente, deve ter uma casa decimal.

Na última f-string, linha 13, utilizamos a expressão {pi:.4f}, para que seja exibido o número com quatro casas decimais. Observe, no console, como ficaram os resultados.

Linhas 20 até 24

No terceiro e último trecho, vamos mostrar como formatar datas em expressões f-string.

Na linha 20, definimos a variável **data** com a data atual, utilizando o método `datetime.now()`.

Na linha 21, criamos uma f-string para exibir o valor da variável **data** sem informar ao Python qual formatação ele deve utilizar `{data}`. Com isso, a data foi impressa no formato padrão: 2020-08-13 10:50:32.262037.

Na linha 22, utilizamos a expressão `{data:%d/%m/%Y}`, que indica que desejamos exibir a data no formato "dia/mês/ano" (13/08/2020). Veja o resultado no console à direita.

Pesquise mais sobre o módulo `datetime` na documentação oficial para descobrir outras maneiras de formatar datas.

Vamos avançar!



Manipulação de strings com Python

Veja agora como acontece a manipulação de strings com Python. Mas antes, para acompanhar o vídeo e poder executar o programa, baixe os arquivos neste [link](#).

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

No script a seguir, executamos uma sequência de métodos de manipulação de string sobre o arquivo texto `exercício_1_modulo_2.txt`, também gravamos o resultado no arquivo `resultado1.txt`.

```
EAD | exercicio_1_modulo_2.py | exercicio_1_modulo_2 | exercicio_1_modulo_2.txt
1 with open("exercicio_1_modulo_2.txt", "r") as arquivo:
2     conteudo = arquivo.read().split(",")
3
4 with open("resultado1.txt", "w") as resultado:
5     for item in conteudo:
6         texto = f"Novo conteúdo {item.strip()}\n"
7         resultado.write(texto)
```

Arquivo texto `exercício_1_modulo_2.txt`.

Identifique qual é o resultado correto contido no arquivo `resultado1.txt`.

A

Novo conteúdo
Banana

B	Uva Laranja
	Novo conteúdo Banana Novo conteúdo Uva Novo conteúdo Laranja
C	Banana Uva Laranja
D	Novo conteúdo Banana Novo conteúdo Uva Novo conteúdo Laranja
E	Banana Uva Laranja Novo conteúdo

Parabéns! A alternativa B está correta.

No script, na linha 2, lemos todo o conteúdo do arquivo utilizando o método `read` e, na sequência, utilizamos o método *split* para quebrar o texto em palavras. Cada palavra é iterada na linha 5 e utilizada para montar a f-string da linha 6. Para cada iteração, o conteúdo da f-string será: “Novo conteúdo” + palavra + “ ”. Cada f-string gerada é escrita no arquivo `resultado1.txt`.

Questão 2

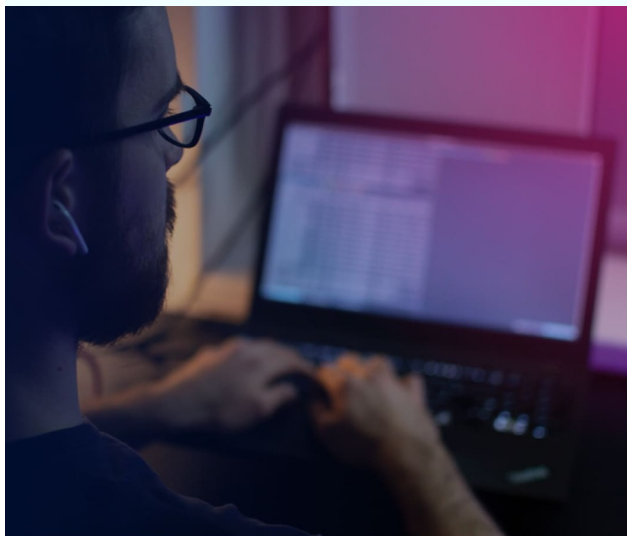
Considere que estamos desenvolvendo um programa e, em determinado ponto, temos uma variável chamada `frase` com a seguinte string: “Ômega;Opala;Monza”.

Qual dos seguintes métodos devemos utilizar para criar uma lista com as palavras Ômega; Opala e Monza?

A	<code>frase.split()</code>
B	<code>frase.strip()</code>
C	<code>frase.split(",")</code>
D	<code>“,”.join(frase)</code>
E	<code>“ ”.join(frase)</code>

Parabéns! A alternativa C está correta.

O método *split* é utilizado para quebrar uma string em uma lista de palavras. O parâmetro passado para esse método é utilizado como separador. Ao executar o comando `frase.split(",")`, criamos a lista ["Ômega", "Opala", "Monza"].



3 - Tratamento de exceções e outras operações

Ao final deste módulo, você será capaz de descrever as exceções na manipulação de arquivos e outras operações.

Conceito de exceções

Tratamento de exceções



Quando trabalhamos com arquivos, é comum encontrarmos alguns problemas, como arquivo inexistente e falta de permissão para escrever em um. A maioria desses problemas só pode ser detectada durante a execução do programa.

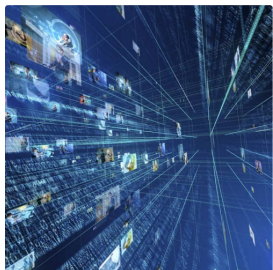
Quando uma falha inesperada ocorre e o interpretador não consegue resolver o problema, dizemos que houve uma exceção.

Nesses casos, precisamos informar ao interpretador como tratar a exceção, para que o programa não seja interrompido.

Se a exceção é um problema inesperado, como tratá-la?

Ao desenvolver um programa, precisamos procurar na documentação da biblioteca, do módulo ou da própria linguagem de programação se as funcionalidades que vamos utilizar têm exceções mapeadas. Essas exceções são problemas que podem ocorrer, e é nossa tarefa tratá-las.

Você deve estar se perguntando: “O que seria ‘tratar uma exceção?’”. Isso nada mais é do que dizer ao Python o que fazer, ou quais instruções executar, quando ele encontrar um problema.



Para ilustrar, observe os seguintes passos:

Quando abrimos um arquivo em modo leitura e esse arquivo não existe, o Python lança uma exceção do tipo `FileNotFoundError`.

Se **não** avisarmos ao Python o que fazer quando isso ocorrer, o programa será interrompido.

Nesse caso, um tratamento para essa exceção poderia ser: exibir um pop-up ao usuário informando que o arquivo não existe.

Veja o que acontece quando uma exceção lançada não é tratada:

Script 22 e saída.

Temos o script 22 e sua saída.

Impressão da string “Abrindo um arquivo”.

Imprimimos, na linha 1, a string “Abrindo um arquivo”. Essa impressão serve para acompanhar a execução do programa no console.

Abertura do arquivo teste.txt no modo leitura.

Abrimos, na linha 3, o arquivo teste.txt no modo leitura. Esse arquivo não existe no diretório modulo3 no qual estamos trabalhando, como podemos observar pela árvore de diretórios à esquerda da figura.

Criação de um contador e aplicação do método *split*.

Recebemos um erro, ao executarmos o programa, destacado em vermelho no console. O nome do erro é FileNotFoundError, e a sua descrição é No such file or directory (em tradução literal, não existe tal arquivo ou diretório), seguido do nome do arquivo que apresentou o erro, teste.txt.

Exceção gerada.

Essa exceção foi gerada, ou lançada, de acordo com o linguajar da computação, pois o Python não sabia qual caminho tomar ao encontrar esse problema. Observe que a linha 5 não foi executada, pois o programa parou sua execução assim que o problema foi encontrado.

Para resolver isso, precisamos tratar a exceção, ou melhor, uma possível exceção. Esse tratamento informa ao Python uma rota alternativa, caso ele encontre um problema.

Para tratar exceções, precisamos “envolver” o trecho de código passível de erro com a cláusula try/except ou try/except/finally. Veremos apenas a cláusula try/except.

Veja a sintaxe a seguir:

```
try:
    # código que pode gerar uma exceção
    ...
except Erro1 as erro:
    # código alternativo caso Erro1
    print(erro)
except Erro2 as erro:
    # código alternativo caso Erro2
    print(erro)
...
```

O código crítico que desejamos executar deve estar no escopo do *try*, enquanto o código alternativo, que será executado em caso de erro, deve estar no escopo do **except**.

Uma mesma operação pode lançar mais de um tipo diferente de exceção, em que, para cada tipo, Erro1 e Erro2, devemos ter uma cláusula **except** específica.

No exemplo da imagem, a exceção está disponível por meio da variável *erro*, de onde podemos extrair mais informações, como veremos a seguir.

Praticamente todas as exceções em Python são herdadas da classe *Exception*, ou seja, ela é uma exceção muito genérica, lançada por diversos tipos de erros diferentes. Quanto mais genérica, mais abrangente é a exceção.

Atenção!

Não é uma boa prática utilizar exceções abrangentes, pois elas podem silenciar erros que não esperamos. O ideal é tratar as exceções utilizando a forma mais específica possível.

Confira algumas exceções específicas relacionadas à manipulação de arquivos e alguns motivos que podem gerar essas exceções:

PermissionError

Lançada quando não temos permissão para realizar uma operação.

FileExistsError

Lançada quando tentamos criar um arquivo ou diretório já existentes.

FileNotFoundError

Lançada quando tentamos abrir um arquivo ou diretório que não existem.

Todas essas exceções herdam da exceção mais abrangente *OSError*, que, por sua vez, herda de *Exception*.

Observe o exemplo a seguir, onde vamos tratar a exceção do exemplo anterior, utilizando a exceção específica mais indicada: *FileNotFoundError*:

Script 23 e saída.

Temos o script 23 e sua saída.

“Envolvemos” o código que pode gerar problema com o try. Para isso, indentamos as linhas 4 e 5.

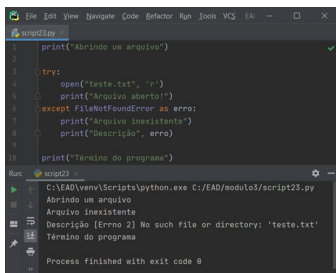
Indentação das linhas 7 e 8.

Precisamos, para tratar o erro, explicitar qual o tipo que vamos tratar. Nesse caso, `FileNotFoundError`, como descrito na linha 6. Indentamos as linhas 7 e 8 para indicar que elas fazem parte do escopo da exceção explicitada na linha 6.

Durante a execução do programa, ao executar a linha 4, o Python encontra um erro, pois tentamos abrir o arquivo `teste.txt` para leitura, mas ele não existe. Como este código está dentro do escopo do try, o interpretador interrompe imediatamente a execução do código contido nesse escopo e inicia a execução do código do `except` correspondente ao erro `FileNotFoundError`. Ou seja, a execução salta da linha 4 para a linha 7.

Na linha 7, imprimimos a mensagem "Arquivo inexistente" e, na linha 8, imprimimos o problema encontrado, disponível na variável `erro`.

Observe a sequência de execução pelas saídas no console:



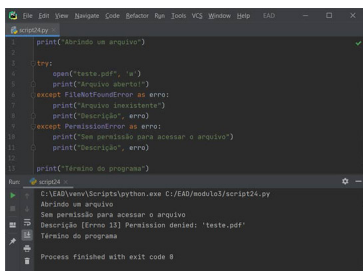
Início da execução do código do `except`.

Saiba que o Python só consegue tratar a exceção caso o erro esteja mapeado em algum `except`. Se o interpretador não encontrar o `except` adequado, será gerado um erro, e o programa será interrompido.

Um problema clássico que ocorre quando lidamos com arquivos é tentar alterar o conteúdo de um arquivo quando ele está aberto em outro programa. No caso do sistema operacional Windows 10, é lançada uma exceção sobre permissão de acesso.

A seguir, vamos criar mais um `except` para tratar o caso de não termos permissão para abrir um arquivo, mostrando o tratamento do problema levantado no parágrafo anterior.

Neste exemplo, tentamos abrir o arquivo `teste.pdf` para **escrita**, linha 4, porém ele já está aberto em outro programa:



Script 24 e sua saída.

Observe que o fluxo de execução do programa saltou da linha 4 para a linha 10. Na linha 10, temos o início do tratamento da exceção `PermissionError`, que foi justamente a exceção lançada pelo Python, impressa pela linha 11, e que pode ser verificada no console.

O Python direciona o fluxo de execução para o trecho onde é realizado o tratamento da exceção lançada.

Vamos explorar mais operações referentes a arquivos e diretórios e mostrar novas exceções que podem ser lançadas quando utilizamos tais operações. Fique atento!

Operações adicionais em arquivos

Além das opções para leitura e escrita em arquivos, o Python disponibiliza um conjunto de operações adicionais, como renomear e apagar arquivo, além de operações em diretórios, como listar arquivos de diretórios, criar diretórios etc.

A partir de agora, apresentaremos algumas dessas operações.

Vamos iniciar pela operação de remover um arquivo, que está disponível por meio da função **remove** do módulo **os** do Python.

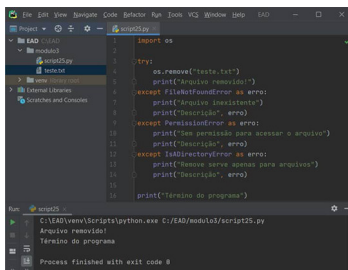
A função **remove** tem a seguinte sintaxe:

```
Python

1 >>> os.remove(caminho)
```

Nesse exemplo, temos o nome do módulo **os**, seguido de um ponto e o nome da função **remove**. Como parâmetro, a função espera o caminho para um **arquivo**. Para remover diretório, devemos utilizar outra função, **rmdir**.

Veja a função **remove** na imagem abaixo. Iniciamos o script com a importação do módulo **os**, na linha 1. Aqui vamos remover o arquivo **teste.txt**, que se encontra no mesmo diretório do nosso script. Observe a árvore de diretórios à esquerda:



Script 25 e sua saída.

Na linha 2, utilizamos a função **remove**, passando como argumento o caminho do arquivo que desejamos remover. Como estamos no mesmo diretório, utilizamos apenas o nome do arquivo. Pronto! Isso é suficiente para remover um arquivo.

Dentre as exceções lançadas ao usar a função **remove**, destacamos as seguintes:

FileNotFoundError

Ocorre quando o arquivo não existe.

PermissionError

Ocorre quando não temos permissão para alterar o arquivo.

PermissionError

Ocorre quando tentamos remover um diretório usando a função `remove`, em vez de `rmdir`.

Observe a saída do console, onde tudo ocorreu conforme esperado e nenhuma exceção foi lançada.

A segunda operação, também muito comum, é a de renomear um arquivo. Essa operação também está disponível no módulo `os`, mas por meio da função `rename`.

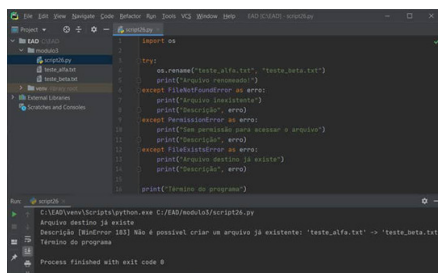
A função `rename` tem a seguinte sintaxe:

```
Python

1 >>> os.rename(origem, destino)
```

Nesse exemplo, temos o nome do módulo `os`, seguido de um ponto e o nome da função `rename`. Como parâmetro, a função espera o caminho para o arquivo que desejamos renomear, **origem**, e o novo nome do arquivo, **destino**.

Veja agora o exemplo em que descrevemos o uso dessa função:



Script 26 e sua saída.

Na linha 1, importamos o módulo `os`, no qual será utilizada a função `rename`.

Na linha 4, chamamos a função `rename` com os parâmetros `teste_alfa.txt` (origem) e `teste_beta.txt` (destino). Caso tudo ocorra bem, ao final da operação, teremos apenas o arquivo destino.

Veja agora algumas exceções que podem ser lançadas quando utilizamos a função `rename`. Não estamos tratando todas as opções possíveis, mas apenas as mais comuns:

FileNotFoundError

Ocorre quando a origem não existe.

FileExistsError

Ocorre quando o arquivo de destino já existe.

PermissionError

Ocorre quando não temos permissão para alterar o arquivo de origem ou para escrita do destino.

Na imagem **Script 26 e sua saída**, veja a árvore de diretórios à esquerda. Temos tanto o arquivo `teste_alfa.txt` quanto o arquivo `teste_beta.txt`.

Observe a execução do script pelo console e veja que ele saltou da linha 4 para a linha 13. Isso ocorreu porque, como o arquivo teste_beta.txt já existia, a exceção `FileExistsError` foi lançada.

Dica

Para os casos em que desejamos renomear sobrescrevendo o arquivo destino, caso ele exista, podemos utilizar a função `replace`, também do módulo `os`.

Manipulação de diretórios

Criando e removendo diretórios

Trabalhar com arquivos significa trabalhar com diretórios. Vejamos as principais funcionalidades relacionadas à manipulação de diretórios em Python começando pela criação e remoção de um diretório.

Para criar um diretório, utilizamos a função `mkdir` do módulo `os`, enquanto, para remover um diretório, utilizamos a função `rmdir`, também do módulo `os`.

A sintaxe dessas duas funções são as seguintes:

Python

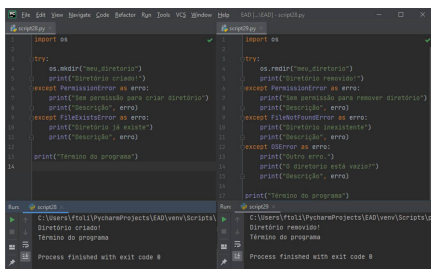
```
1 >>> os.mkdir(caminho)
```

Python

```
1 >>> os.rmdir(caminho)
```

Nesse exemplo, temos o nome do módulo `os`, seguido de um ponto e o nome da função `mkdir` ou `rmdir`. Como parâmetro, a função espera o caminho para o diretório.

Veja agora como utilizamos essas duas funções:



Scripts 28 e 29 e suas respectivas saídas.

No script 28, importamos o módulo **os** na linha 1 e, na linha 4, utilizamos a função `mkdir("meu_diretorio")`. O diretório `meu_diretorio` foi criado na mesma pasta onde o script28 se encontra. Considere as seguintes condições:

- Caso não tenhamos permissão para criar o diretório, será lançada a exceção **PermissionError**.
- Caso o diretório já exista, a exceção **FileExistsError** é lançada.

No script29, na linha 4, utilizamos a função `rmdir` para remover o diretório `meu_diretorio`. Considere as seguintes condições:

- Caso não tenhamos permissão para remover o diretório, será lançada a exceção **PermissionError**.
- Caso o diretório não exista, a exceção **FileNotFoundError** é lançada.

Para os casos em que o diretório a ser removido não esteja vazio, será lançada a exceção `OSError`. Essa exceção é mais abrangente.

Não temos como garantir, a princípio, que a exceção lançada ocorre especificamente pelo fato de o diretório não estar vazio.

Nessas situações, precisamos analisar mais o erro, principalmente o seu número, para verificar o que realmente aconteceu.

O número do erro está disponível no atributo `errno` do objeto erro.

Os códigos dos possíveis erros estão no módulo `errno` do Python e podem ser utilizados no tratamento das exceções para descobrir o que realmente deu errado.

Veja esse problema mais de perto:

Script 30 e saída.

Temos o script 30, sua saída e módulo `errno.py`.

Importação dos módulos `os` e `errno`.

Importamos, à esquerda, no script30, os módulos `os` e `errno` nas linhas 1 e 2.

Tentativa de remoção do diretório meu_diretorio.

Tentamos remover, na linha 5, o diretório meu_diretorio utilizando a função rmdir. Como o diretório não está vazio, a exceção OSError é lançada, e a execução do programa salta para a linha 8.

Impressão do número do erro.

Imprimimos, na linha 8, o número do erro por meio do atributo errno da variável erro. Observe que o valor impresso foi 41. O erro 41 faz parte da numeração interna de erros do Python, que pode ser verificado no modulo errno à direita da imagem, mapeado como ENOTEMPTY (não vazio).

Comparação entre o erro gerado e o código do erro.

Comparamos, na linha 9, o erro gerado pelo programa com o código do erro ENOTEMPTY (erro 41). Caso o resultado da comparação seja verdadeiro, teremos certeza de que o erro ocorreu, pois o diretório não está vazio. Caso contrário, precisaremos analisá-lo novamente. O WinError 145 é o erro nativo que o Windows retornou. Esse erro foi mapeado pelo Python para o erro 41.

Como a exceção OSError é mais abrangente que as outras exceções que estudamos, ela deve ficar por último. Caso contrário, nunca alcançaremos as exceções mais específicas.

Listando conteúdo de diretórios

Outra tarefa muito comum quando estamos tratando com arquivos é listar os arquivos presentes em um diretório.

Para isso, podemos utilizar a função scandir do módulo os. Sua sintaxe é a seguinte:

```
Python
```

```
1 >>> os.scandir(caminho)
```


Nesse exemplo, temos o nome do módulo **os**, seguido de um ponto e o nome da função `scandir`. Como parâmetro, a função espera o caminho para o **diretório**.

Como resultado, teremos um iterável (iterator) que retorna objetos do tipo `os.DirEntry`, que podem ser arquivos ou diretórios. Dentre os atributos e métodos desse tipo de objetos, destacamos:

Name

Nome do diretório ou arquivo.

Path

Caminho completo do diretório ou arquivo.

is_dir()

Retorna verdadeiro se o objeto é um diretório.

is_file()

Retorna verdadeiro se o objeto é um arquivo.

stat()

Retorna alguns atributos do arquivo ou diretório, como tamanho.

Agora veja como utilizar essa função:

Script 31 e saída.

Temos o script 31 e sua saída. Vamos percorrer os arquivos e diretórios da pasta `meu_diretorio`. A árvore de diretórios pode ser verificada à esquerda da imagem.

Importação dos módulos os.

Importamos, na linha 1, o módulo os, onde se encontra a função scandir.

Aplicação da função scandir.

Utilizamos, na linha 4, a função scandir utilizando o diretório "meu_diretorio" como argumento. Armazenamos o retorno dessa função na variável entradas.

Iteração de cada entrada.

Iteramos, na linha 6, cada entrada e, da linha 7 a 13, imprimimos algumas de suas propriedades.

Saída no console à direita.

Observe a saída no console à direita.



Manipulação de arquivos e diretórios com Python

Confira a explicação sobre manipulação de arquivos e diretórios com Python.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Quando realizamos operações com arquivos, é uma boa prática tratar as exceções que podem ocorrer. Qual é a sequência de tratamento de exceções que devemos utilizar para garantir que as exceções que podem ser lançadas ao remover um arquivo utilizando a função `os.remove()` sejam capturadas e tratadas corretamente?

A Exception - OSError - FileNotFoundError - PermissionError - IsADirectoryError

B FileNotFoundError - PermissionError - Exception - OSError - IsADirectoryError

C OSError - FileNotFoundError - PermissionError - IsADirectoryError - Exception

D FileNotFoundError - PermissionError - IsADirectoryError - OSError - Exception

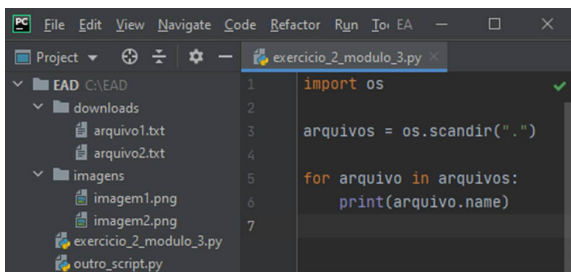
E FileNotFoundError - OSError - IsADirectoryError - PermissionError - Exception

Parabéns! A alternativa D está correta.

A função `os.remove()` pode lançar as exceções `FileNotFoundError`, `PermissionError` e `IsADirectoryError`. Para conseguir tratar essas exceções, isso precisa ser feito antes das exceções mais abrangentes `Exception` e `OSError`. A opção D é a única na qual essas três exceções estão antes das mais abrangentes.

Questão 2

No script a seguir, utilizamos o método `scandir` para mostrar os arquivos do diretório explicitado na linha 3.



Aplicação do método scandir.

Considere a árvore de diretórios à esquerda da imagem e assinale a resposta que contém a saída do programa.

A arquivo1.txt - arquivo2.txt - imagem1.png - imagem2.png - exercicio_2_modulo_3.py - outro_script.py

B downloads - imagens - outro_script.py

C downloads - imagens - exercicio_2_modulo_3.py - outro_script.py

D downloads - imagens

E downloads - outro_script.py

Parabéns! A alternativa C está correta.

A função scandir retorna todo o conteúdo do diretório passado como argumento, incluindo arquivos e subdiretórios. Ela não entra em subdiretórios de forma recursiva.

Considerações finais

Neste conteúdo, visitamos as principais ações que podemos realizar na manipulação de arquivos e diretórios.

Vimos como abrir um arquivo, ler seu conteúdo e escrever novos conteúdos.

Aproveitamos para mostrar funções de manipulação de strings, que são normalmente relacionadas ao tratamento de arquivos, como quebrar linhas em palavras e juntar listas de strings a um conector em particular.

Além disso, entendemos as principais exceções geradas quando estamos trabalhando com arquivos, de forma a tratá-las e garantir o correto funcionamento de nossos programas.

Para encerrar, ouça um resumo sobre os principais assuntos deste conteúdo.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



Referências

Python. **Python Software Foundation**. Consultado em meio eletrônico em: 10 ago. 2020.

Explore +

Confira agora o que separamos especialmente para você!

Além de manipular arquivos de texto, o Python permite a leitura e escrita de arquivos binários. O tratamento de arquivos comprimidos é um exemplo disso. O Python possui um módulo interno chamado gzip que permite a leitura e escrita desse tipo de arquivo. Pesquise sobre o módulo gzip e verifique os comandos de manipulação existentes.

Outro tipo de arquivo binário é a imagem. Apesar de o Python não ter um módulo interno que suporte manipulação de imagens, existem bibliotecas gratuitas que dão esse suporte, como Pillow e OpenCV. Pesquise sobre essas bibliotecas e verifique os recursos disponíveis para a manipulação de imagens.