

Linguagem de Programação

Extra: Arquivos



Nossos **objetivos** nesta aula são:

- Explicar o que é persistir dados e conceito de arquivos;
- Criar, abrir, escrever e consultar dados persistidos em arquivos de texto por meio de programas construídos em Python.



MOTIVAÇÃO

Até este ponto da disciplina, nossos programas já utilizaram muitos recursos e estruturas úteis para receber dados, processá-los e gerar saídas. Porém, ao desligar o computador ou simplesmente fechar o programa, essas saídas são perdidas! As entradas também são descartadas, e caso seja necessário processá-las novamente teremos que digitá-las, o que pode ser um problema quando a entrada é composta por muitos dados.

Então, o que queremos é preservar as saídas em um espaço de memória não volátil, ou seja, queremos *persistir* os dados em outro local que não seja a memória RAM, que é onde estão as variáveis e os códigos dos programas. Exemplos de memórias não voláteis são HD, SSD, Pen Drive, SD Card ou mesmo em servidores na nuvem.

Como mencionado, também é útil conseguir as entradas para os programas de alguma fonte que não seja um teclado, pois a necessidade de digitar valores alguma vezes pode ser impraticável, dependendo do tamanho da entrada e da frequência com que o programa precise lê-las.

Neste documento veremos como fazer isso em Python, com o uso de arquivos tanto para persistir os dados (guardando as saídas do programa), quanto para facilitar a inserção de dados (lendo as entradas do programa).

O QUE É UM ARQUIVO?

Simplificadamente, um arquivo é uma área de memória onde podemos realizar a leitura e a escrita de dados. Essa área geralmente está localizada em um dispositivo que permite a persistência dos dados, que é justamente o que não ocorre na memória RAM. A persistência consiste em garantir que, mesmo ao cortar o fluxo de energia do computador ou fechar o programa, os dados permanecerão guardados e poderão ser acessados depois.



Como o gerenciamento da área de memória onde serão guardados os dados do arquivo é de responsabilidade do sistema operacional, não é necessário que o programador se encarregue desse nível de detalhes para criar programas que usem arquivos, pelo menos não em Python.

Porém, existem procedimentos básicos que o programador deve conhecer para trabalhar com arquivos. Por exemplo, é necessário saber o local onde o arquivo está para ser consultado (leitura) e definir o nome e o local onde será guardado ou alterado (escrita).

Existem arquivos com diversas finalidades. Em seu smartphone provavelmente há fotos, vídeos, músicas e textos. Eles são arquivos! Porém de tipos diferentes e por isso precisam de programas diferentes para serem abertos e lidos, afinal cada tipo de arquivo possui peculiaridades que precisam ser consideradas, de modo a serem corretamente interpretados pelo computador. Em nosso caso, estamos interessados somente em arquivos de texto, pois têm manipulação mais simples e não demandam muito conhecimento prévio.

ABERTURA E CRIAÇÃO DE ARQUIVOS

Supondo que o arquivo exista, para abri-lo é preciso especificar seu nome e o diretório onde está, além das operações que pretendemos realizar, que são essencialmente escrita e leitura de dados.

Para abrir arquivos em programas feitos em Python podemos usar a função `open()`. Essa função exige duas strings como parâmetros: *nome do arquivo* e *o modo de abertura*.

Em relação ao *nome do arquivo*, caso o arquivo esteja no mesmo diretório do programa que irá usá-lo, basta escrever somente seu nome, por exemplo, `'compras.txt'`. Caso o arquivo esteja em outro local, pode-se escrever o caminho completo e o nome, por exemplo, `'c:\meus arquivos\compras.txt'`.

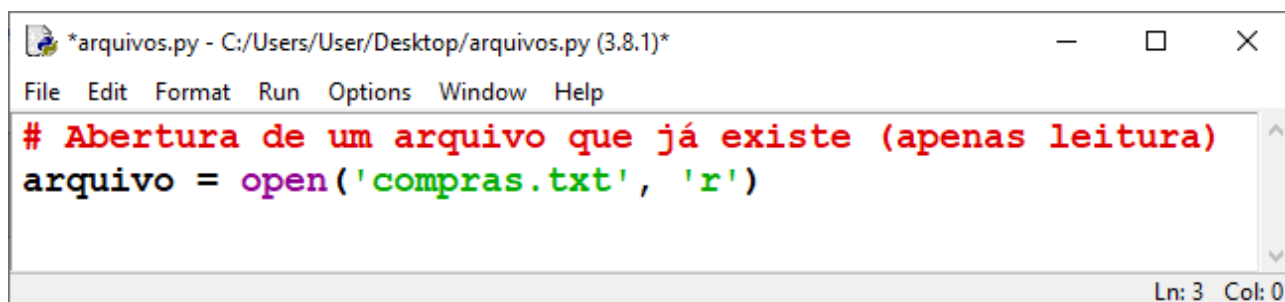
O *modo de abertura* define as operações que serão permitidas no arquivo aberto. Repare que dependendo de fatores relacionados às permissões de acesso configuradas no sistema operacional, um arquivo pode ser aberto de um modo e não de outro. Vide a Tabela 1 em que constam alguns modos de abertura de arquivos.

MODO DE ABERTURA	OPERAÇÕES PERMITIDAS
r (read)	Leitura (o arquivo deve existir).
w (write)	Escrita (apaga o conteúdo pré-existente ou cria um novo arquivo caso ele não exista).
a (append)	Escrita (preserva o conteúdo pré-existente adicionando novos dados no final do arquivo).
r+ (read + write)	Leitura + escrita (preserva o conteúdo pré-existente)
w+ (write + read)	Escrita + leitura (apaga o conteúdo pré-existente ou cria um novo arquivo caso ele não exista)
a+ (append + read)	Escrita + leitura (preserva o conteúdo pré-existente adicionando novos dados no final do arquivo).

x	Abre o arquivo apenas para criação (erro caso o arquivo exista).
t (text)	Abertura em modo texto (é o padrão, não precisa ser especificado).
b (binary)	Abertura em modo binário (não será abordado).

Tabela 1 - Modos de abertura de arquivos.

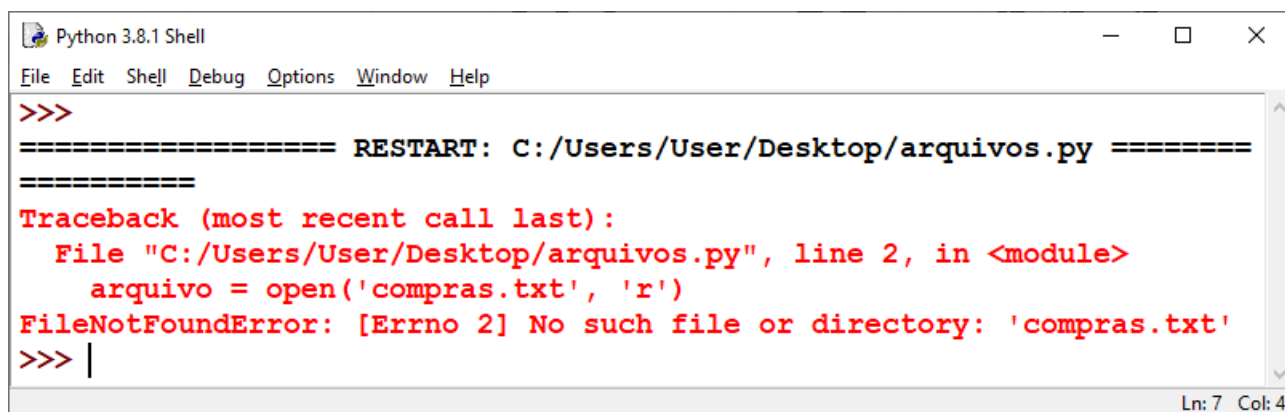
A sintaxe da função `open()`, juntamente com um exemplo de uso, está ilustrada na Figura 1. Note que é necessário guardar o valor retornado pela função em uma variável que, convenientemente, denominados de `arquivo`.



```
*arquivos.py - C:/Users/User/Desktop/arquivos.py (3.8.1)*
File Edit Format Run Options Window Help
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
Ln: 3 Col: 0
```

Figura 1 - Exemplo de utilização da função `open()`.

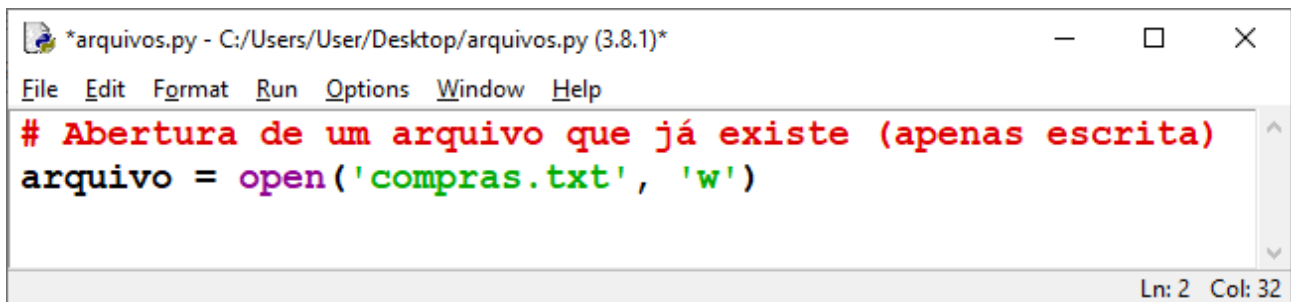
Note que se executarmos esse programa e o arquivo `compras.txt` não estiver no diretório em que guardamos nosso código fonte em Python, ocorrerá um erro, indicando exatamente que o arquivo não foi encontrado, conforme Figura 2.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
>>>
===== RESTART: C:/Users/User/Desktop/arquivos.py =====
Traceback (most recent call last):
  File "C:/Users/User/Desktop/arquivos.py", line 2, in <module>
    arquivo = open('compras.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'compras.txt'
>>> |
Ln: 7 Col: 4
```

Figura 2 - Execução da codificação da Figura 1.

Porém, se o arquivo fosse aberto com um modo diferente de `'r'` este erro não aconteceria, como ilustrado na Figura 3.



```
*arquivos.py - C:/Users/User/Desktop/arquivos.py (3.8.1)*
File Edit Format Run Options Window Help
# Abertura de um arquivo que já existe (apenas escrita)
arquivo = open('compras.txt', 'w')
Ln: 2 Col: 32
```

Figura 3 - Alteração do modo de abertura do arquivo.

Como descrito na Tabela 1, isso ocorre porque caso seja tentado abrir um arquivo que não exista com os modos `'w'` ou `'a'` ele será criado. Tenha cuidado, um arquivo que exista, se aberto com `'w'`, terá todo seu conteúdo sobrescrito. Pronto! Aprendemos a criar arquivos.

Importante: note que o **nome externo** do arquivo que abrimos é `compras.txt`, porém o **nome interno**, que é o que usamos para referenciá-lo dentro do nosso programa, é sempre aquele da variável que recebeu o retorno da função `open()`, neste caso denominada `arquivo`.

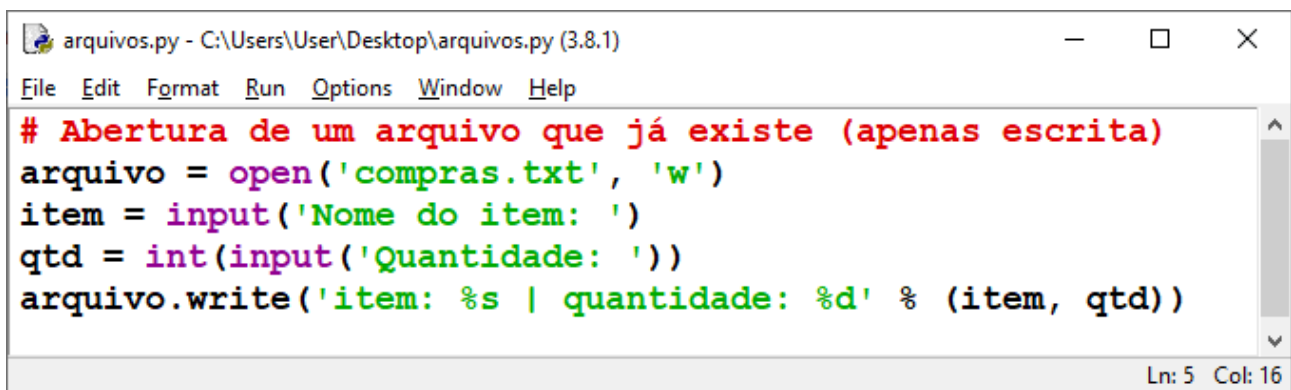
ESCRITA E LEITURA DE DADOS EM ARQUIVOS

Provavelmente, um programa que usa arquivos fará diversas operações de *escrita* e *leitura* de dados, por isso veremos agora como escrever e ler dados em arquivos abertos.

ESCRITA DE DADOS

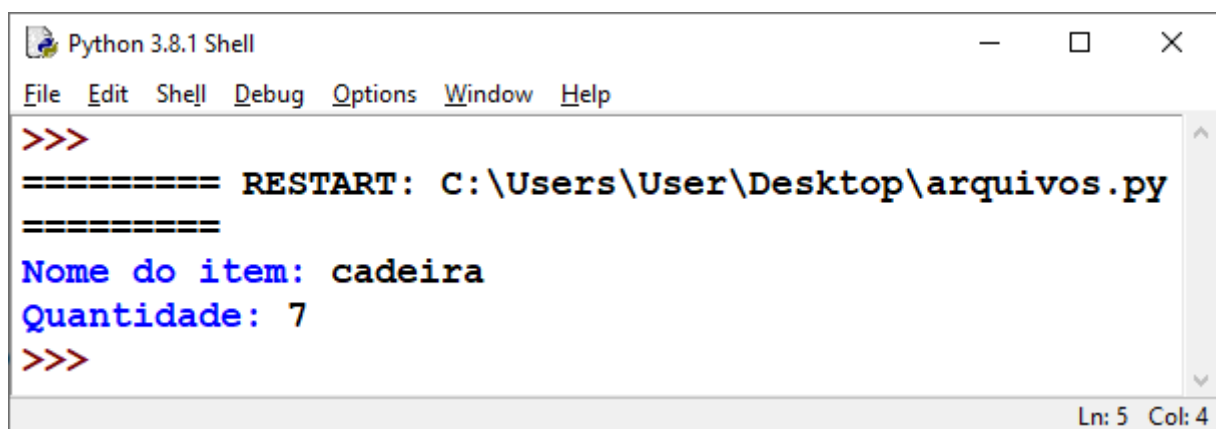
Para escrever dados em um arquivo usaremos o método `write()`, que é semelhante à função `print()`, porém ao invés de exibir saídas na tela, escreverá saídas no arquivo especificado. A sintaxe é `arquivo.write('dados')`, note que o parâmetro de `write()` é uma *string*.

Um exemplo: suponha que você precise criar um programa que solicite ao usuário o nome de um item, a quantidade que precisa ser comprada desse item e, ao final, escreva esses dados em um arquivo. Veja a codificação da Figura 4, que propõe uma solução para o problema e a respectiva tela de execução na Figura 5.



```
# Abertura de um arquivo que já existe (apenas escrita)
arquivo = open('compras.txt', 'w')
item = input('Nome do item: ')
qtd = int(input('Quantidade: '))
arquivo.write('item: %s | quantidade: %d' % (item, qtd))
```

Figura 4 - Primeira proposta de solução para o problema da compra.



```
>>>
===== RESTART: C:\Users\User\Desktop\arquivos.py
=====
Nome do item: cadeira
Quantidade: 7
>>>
```

Figura 5 - Execução da codificação da Figura 4.

Ótimo! Execute o programa e veja o que ocorre. Sem nenhum problema de execução, aparentemente. Notou algo estranho? Ainda não? O arquivo foi gerado no mesmo diretório em que seu código fonte está armazenado, abra-o e veja que `compras.txt` continua em branco!

Para entender a causa desse comportamento inesperado é necessário explicar o conceito de *buffer*. O *buffer* é um espaço de memória usado para armazenar dados temporariamente, antes de serem persistidos no arquivo determinado, é comum que esse espaço temporário seja definido na própria memória RAM.

Existem algumas razões para o *buffer* existir, uma delas é que escrever dados diretamente em um arquivo, normalmente persistido em um HD, consumiria muito tempo. Imagine se a cada caractere que precisássemos persistir fosse necessário colocar os bits correspondentes no HD que, é bom lembrar, se trata de um dispositivo mecânico que precisa posicionar a cabeça de leitura/gravação em local específico de um disco de metal e magnetizá-lo.

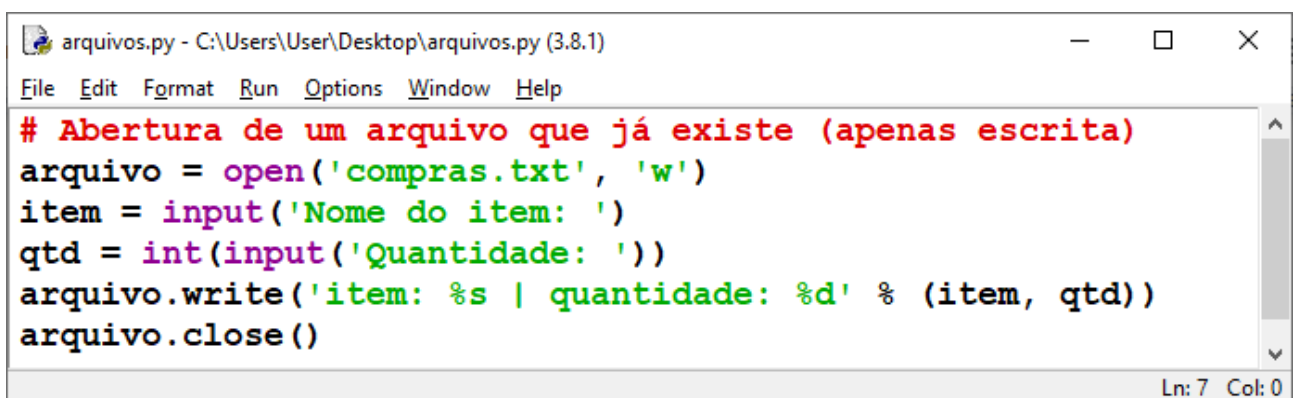
Por isso, quando um comando de escrita é executado, os dados não são escritos diretamente no local definitivo, ou seja, não são persistidos de imediato. Primeiramente os dados são

armazenados no *buffer* de memória e depois "descarregados" no arquivo. A questão é: quando os dados serão retirados do *buffer* e persistidos no arquivo?

Basicamente, o *buffer* é *esvaziado* em duas situações: (I) quando está cheio, pois há um limite de espaço destinado ao *buffer*; (II) quando o programa solicita que o arquivo aberto seja fechado.

Portanto, é uma boa prática de programação sempre fechar os arquivos abertos logo após o encerramento da escrita de dados. Isso garante que todos os dados do *buffer* sejam descarregados no arquivo e, de fato, persistidos.

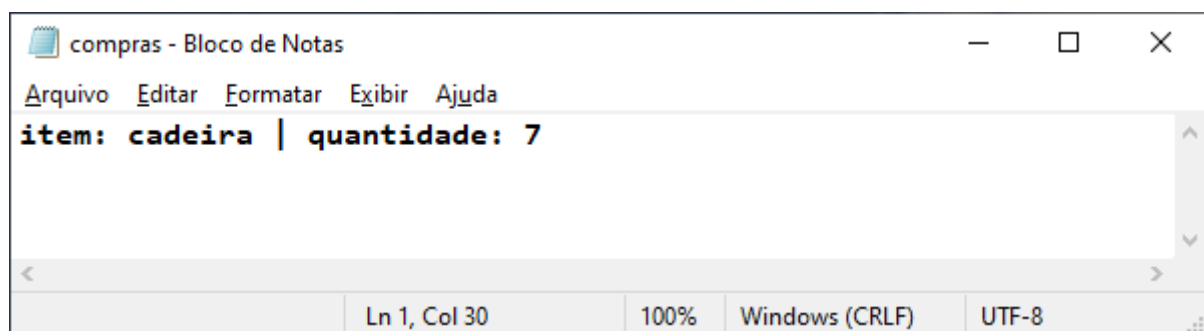
Logo, para resolver o problema na primeira proposta de solução, basta usar o método `close()`, veja na Figura 6 a segunda proposta de solução e na Figura 7 o arquivo gerado.

A screenshot of a text editor window titled "arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The code is as follows:

```
# Abertura de um arquivo que já existe (apenas escrita)
arquivo = open('compras.txt', 'w')
item = input('Nome do item: ')
qtd = int(input('Quantidade: '))
arquivo.write('item: %s | quantidade: %d' % (item, qtd))
arquivo.close()
```

The status bar at the bottom right shows "Ln: 7 Col: 0".

Figura 6 - Segunda proposta de solução para o problema da compra.

A screenshot of a text editor window titled "compras - Bloco de Notas". The window has a menu bar with "Arquivo", "Editar", "Formatar", "Exibir", and "Ajuda". The text content is:

```
item: cadeira | quantidade: 7
```

The status bar at the bottom shows "Ln 1, Col 30", "100%", "Windows (CRLF)", and "UTF-8".

Figura 7 - Arquivo gerado com a codificação da Figura 6.

Podemos incrementar nosso programa e montar uma lista de compras com diversos itens. Para isso basta acrescentar um laço de repetição e poucas modificações, conforme a Figura 8. Veja na Figura 9 o programa modificado em execução.

```
# Abertura de um arquivo que já existe (apenas escrita)
arquivo = open('compras.txt', 'w')
item = input('Nome do item: ')
while item != '':
    qtd = int(input('Quantidade: '))
    arquivo.write('item: %s | quantidade: %d' % (item, qtd))
    item = input('Nome do item: ')
arquivo.close()
```

Figura 8 - Versão da solução que aceita diversos itens para a compra.

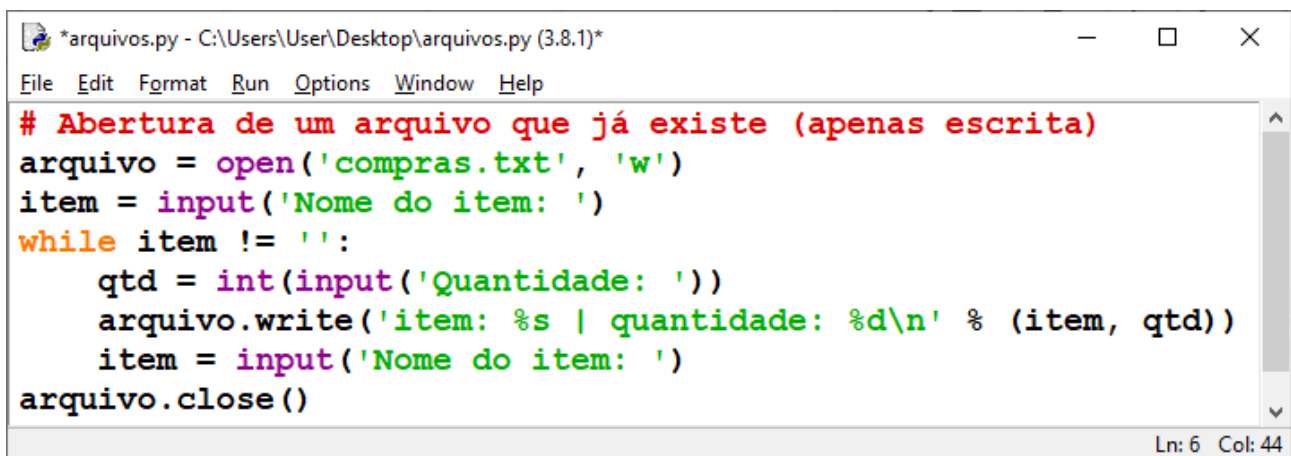
```
>>>
===== RESTART: C:\Users\User\Desktop\arquivos.py ==
=====
Nome do item: cadeira
Quantidade: 7
Nome do item: mesa
Quantidade: 2
Nome do item: lousa
Quantidade: 1
Nome do item:
>>>
```

Figura 9 - Execução da codificação da Figura 8.

É importante notar que, ao contrário da função `print()`, o método `write()` não quebra a linha automaticamente após cada execução, veja o arquivo gerado na Figura 10. Por isso é necessário colocar manualmente no final do parâmetro *string* o caractere `'\n'`. Veja a modificação do código na Figura 11 e o novo arquivo gerado na Figura 12.

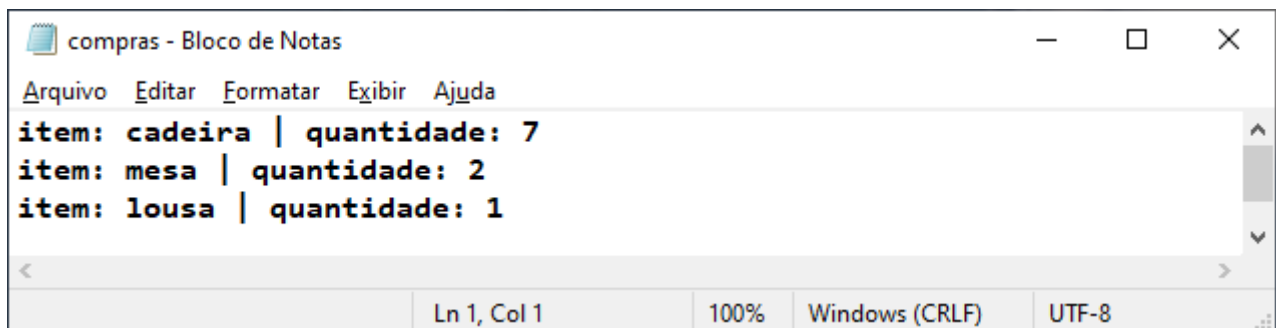
```
item: cadeira | quantidade: 7item: mesa | quantidade: 2item: lousa | quantidade: 1
```

Figura 10 - Arquivo gerado com a codificação da Figura 8.



```
# Abertura de um arquivo que já existe (apenas escrita)
arquivo = open('compras.txt', 'w')
item = input('Nome do item: ')
while item != '':
    qtd = int(input('Quantidade: '))
    arquivo.write('item: %s | quantidade: %d\n' % (item, qtd))
    item = input('Nome do item: ')
arquivo.close()
```

Figura 11 - Alteração para quebrar a linha após cada execução do método `write()`.



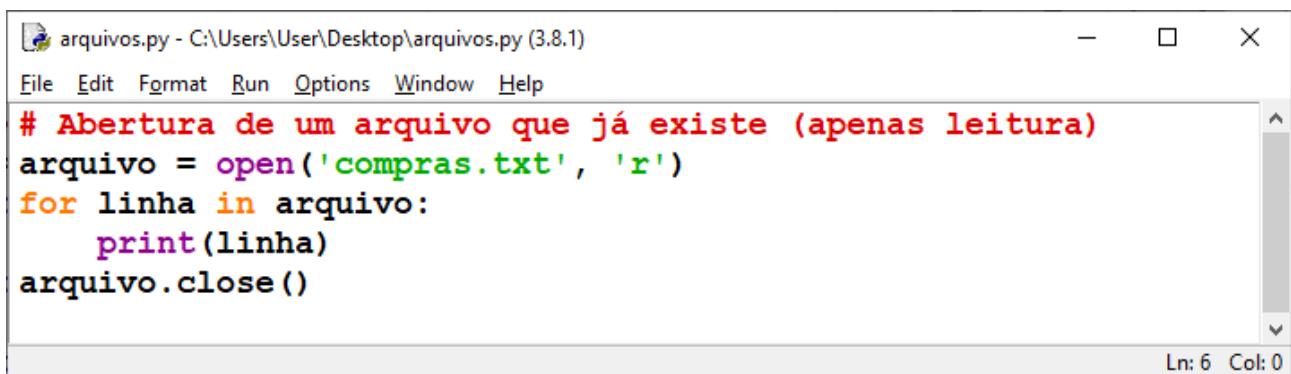
```
item: cadeira | quantidade: 7
item: mesa | quantidade: 2
item: lousa | quantidade: 1
```

Figura 12 - Arquivo gerado com a codificação da Figura 11.

Experimente executar o mesmo programa novamente inserindo outros itens e, em seguida, abra o arquivo gerado. Note que o conteúdo foi sobrescrito. Para manter os dados pré-existent e acrescentar outros, mude o modo de abertura para `'a'`, assim os dados serão anexados ao final do arquivo.

LEITURA DE DADOS

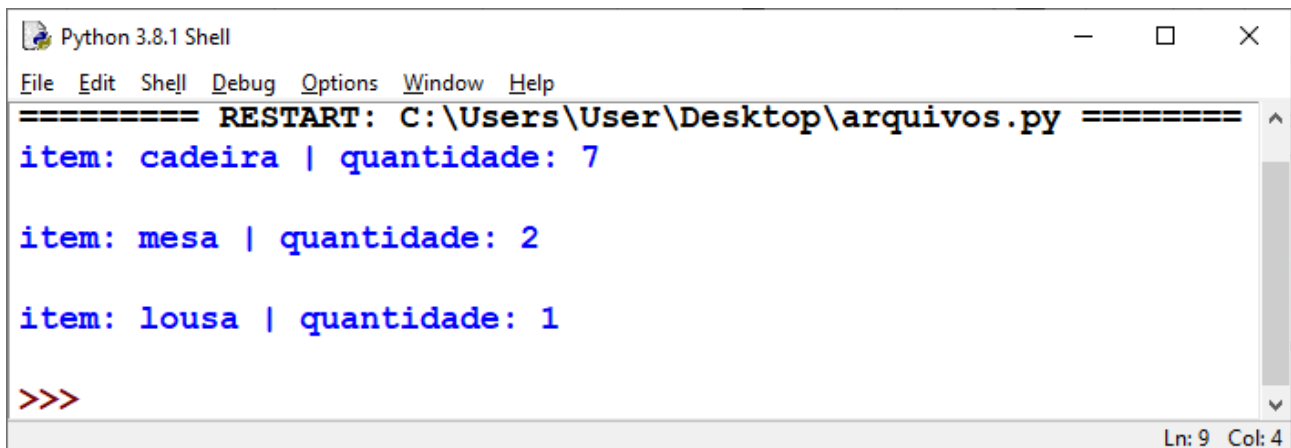
Com um arquivo criado, podemos abrí-lo, ler seus dados e exibi-los na tela. Em Python, é possível trabalhar com arquivos de texto como se fossem listas de *strings*, onde cada linha do arquivo é um item da lista. Desta forma podemos ler todas as linhas de um arquivo simplesmente usando um laço `for`, como ilustrado na Figura 13.



```
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
for linha in arquivo:
    print(linha)
arquivo.close()
```

Figura 13 - Leitura dos dados de um arquivo apenas com um laço for.

Ao executar o programa, cada linha do arquivo é lida e exibida na tela, como pode ser visto na Figura 14.



```
==== RESTART: C:\Users\User\Desktop\arquivos.py =====
item: cadeira | quantidade: 7

item: mesa | quantidade: 2

item: lousa | quantidade: 1

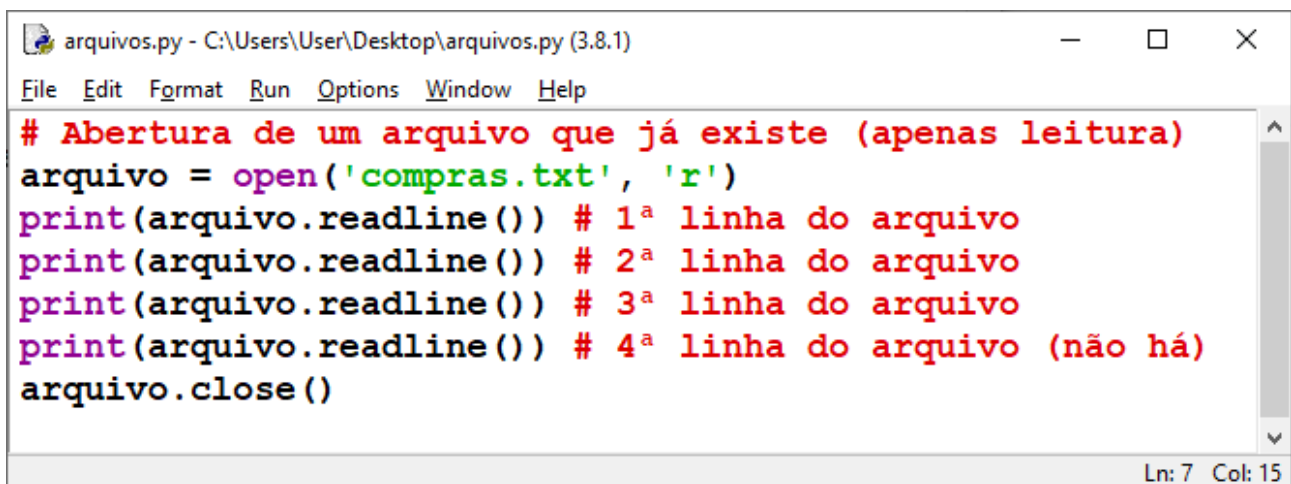
>>>
```

Figura 14 - Execução da codificação da Figura 13.

Caso tenha ficado em dúvida sobre a razão de ter uma linha em branco após toda linha de texto exibida: (I) a função `print()`, por padrão, gera uma quebra de linha após cada execução; (II) cada uma das linhas do arquivo já contém uma quebra de linha explicitamente colocada no final com o caractere `'\n'`. Logo, por causa de (I) e (II) na tela são exibidas duas quebras de linha.

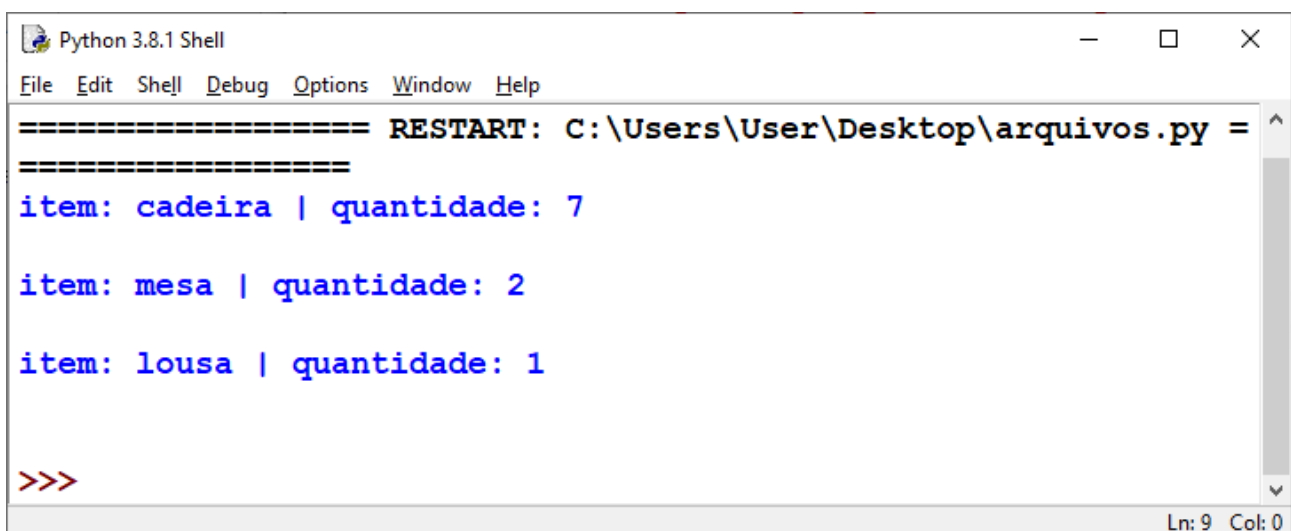
• Leitura de uma linha por vez

Há outras formas de ler um arquivo. Uma forma é usando o método `readline()`, que lê apenas uma linha do arquivo cada vez que é executado e retorna uma *string* com os dados da linha lida. Veja a Figura 15 com a codificação e a Figura 16 com a exibição dos dados lidos.



```
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
print(arquivo.readline()) # 1ª linha do arquivo
print(arquivo.readline()) # 2ª linha do arquivo
print(arquivo.readline()) # 3ª linha do arquivo
print(arquivo.readline()) # 4ª linha do arquivo (não há)
arquivo.close()
```

Figura 15 - Leitura dos dados de um arquivo com o método `readline()`.



```
===== RESTART: C:\Users\User\Desktop\arquivos.py =
=====
item: cadeira | quantidade: 7

item: mesa | quantidade: 2

item: lousa | quantidade: 1

>>>
```

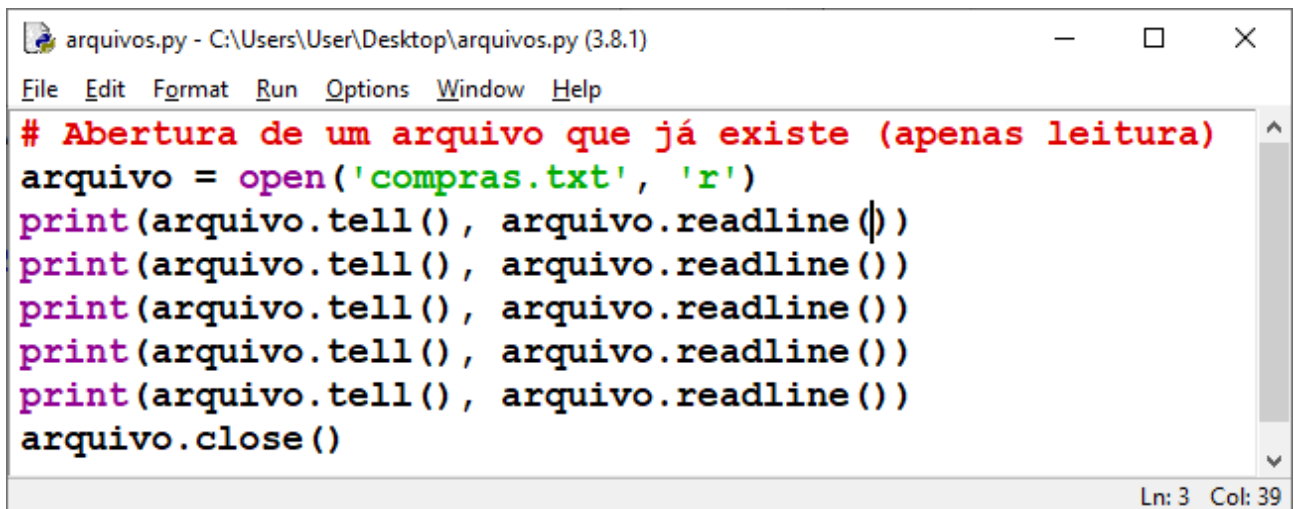
Figura 16 - Execução da codificação da Figura 15.

Repare que a quarto `print()` não exibiu nada além de uma quebra de linha, isso ocorreu porque o arquivo já não continha mais linhas para que `readline()` retornasse, neste caso o método retornará uma *string* vazia.

Cabe uma explicação adicional neste ponto. Ao abrir um arquivo para leitura, o Python estabelece um marcador com a posição atual do arquivo (próximo caractere que será lido), inicialmente este marcador indica a posição zero. Depois de usar o método `readline()`, o marcador passará a indicar a posição do primeiro caractere da próxima linha.

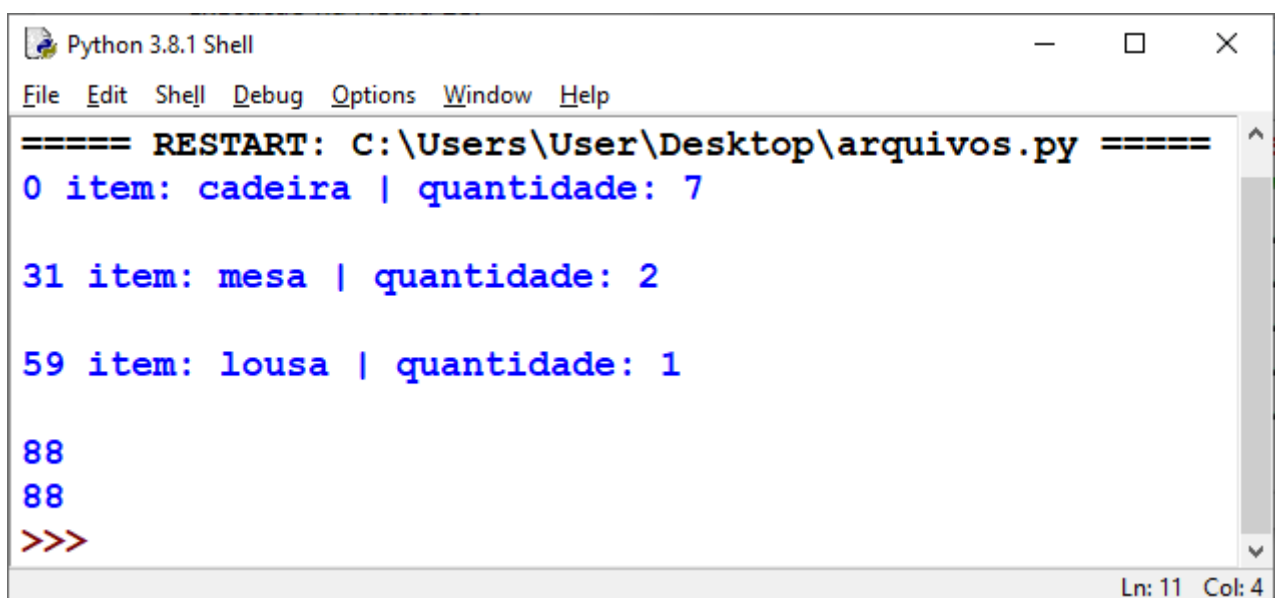
Em nosso arquivo de exemplo, a primeira linha possui 31 caracteres: 29 (letras, números e espaços) + 2 (quebra de linha '`\n`', que dentro de arquivos em Windows se transforma em dois

caracteres¹). Portanto, após a execução do primeiro `readline()` o marcador indicará a posição 31 (lembre-se que todo arquivo começa na posição zero). Para saber a posição atual do arquivo use o método `tell()`, conforme a codificação ilustrada na Figura 17 e a respectiva execução na Figura 18.



```
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
print(arquivo.tell(), arquivo.readline())
print(arquivo.tell(), arquivo.readline())
print(arquivo.tell(), arquivo.readline())
print(arquivo.tell(), arquivo.readline())
print(arquivo.tell(), arquivo.readline())
arquivo.close()
```

Figura 17 - Leitura dos dados de um arquivo e exibição do valor do ponteiro de arquivo com `tell()`.



```
==== RESTART: C:\Users\User\Desktop\arquivos.py ====
0 item: cadeira | quantidade: 7

31 item: mesa | quantidade: 2

59 item: lousa | quantidade: 1

88
88
>>>
```

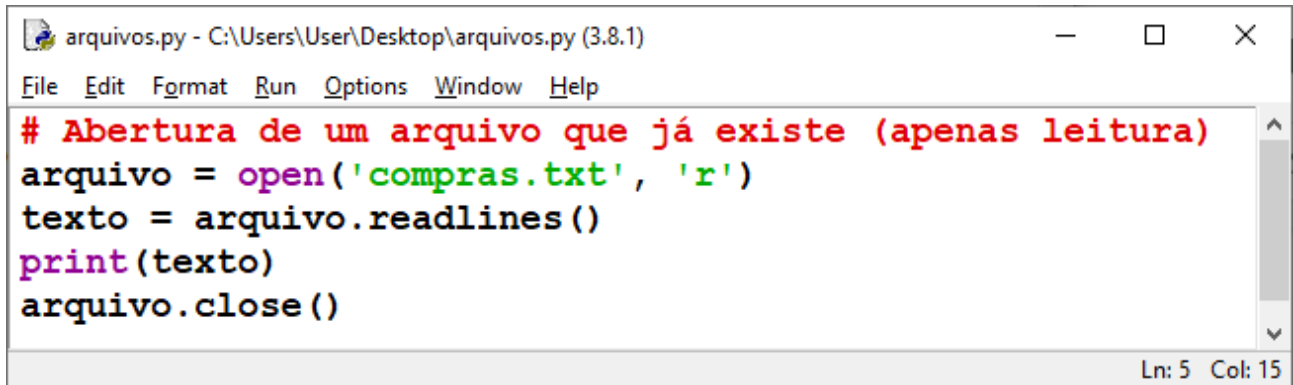
Figura 18 - Execução da codificação da Figura 17.

Obs.: Note que os últimos dois valores são idênticos. Isso ocorreu porque o marcador já estava no fim do arquivo e, portanto, não poderia avançar mais.

¹ Quando os dados são persistidos em um arquivo de texto, a representação de quebra de linha depende do padrão do sistema operacional. Ao fazer a leitura dos dados para Python, a quebra de linha volta ao caractere '`\n`' novamente.

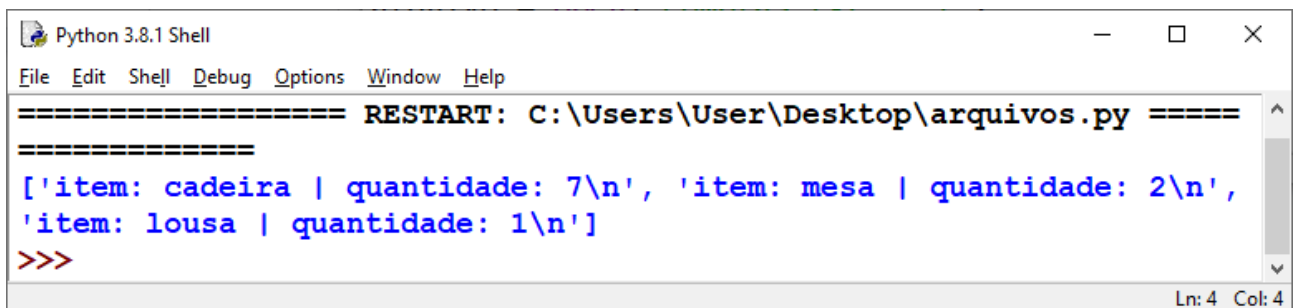
- **Leitura de todo o arquivo**

Também existe o método `readlines()`, que lê o arquivo inteiro e retorna uma lista de *strings*. Veja um exemplo de codificação na Figura 19 e a respectiva execução na Figura 20.



```
arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)
File Edit Format Run Options Window Help
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
texto = arquivo.readlines()
print(texto)
arquivo.close()
Ln: 5 Col: 15
```

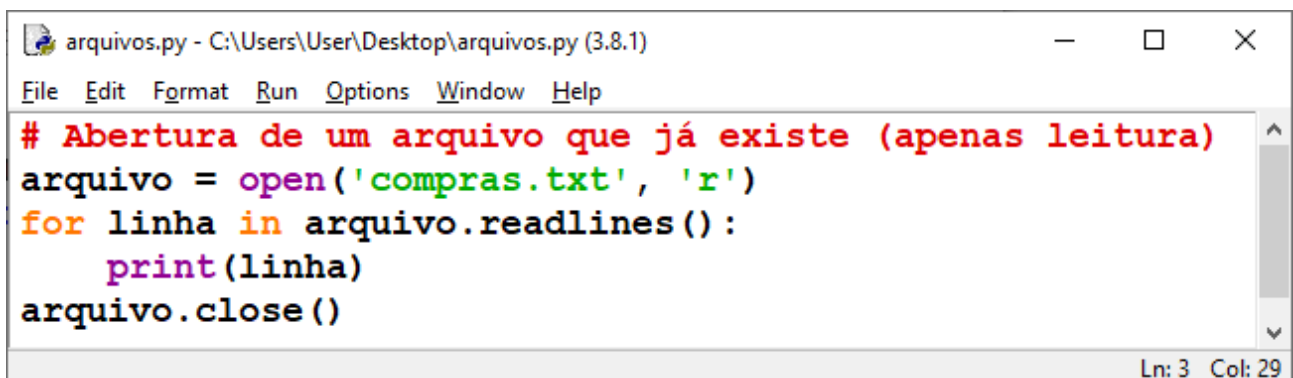
Figura 19 - Leitura dos dados de um arquivo com o método `readlines()`.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
===== RESTART: C:\Users\User\Desktop\arquivos.py =====
=====
['item: cadeira | quantidade: 7\n', 'item: mesa | quantidade: 2\n',
'item: lousa | quantidade: 1\n']
>>>
Ln: 4 Col: 4
```

Figura 20 - Execução da codificação da Figura 19.

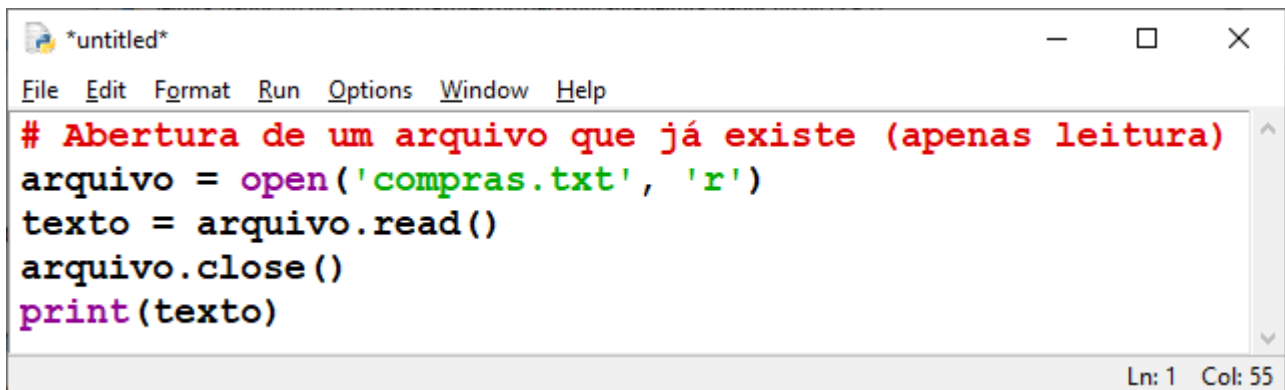
Para que os dados sejam apresentados linha a linha, é possível usar a mesma lógica apresentada na leitura usando apenas o laço `for`, iterando item por item da lista de *strings*. Vide Figura 21.



```
arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)
File Edit Format Run Options Window Help
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
for linha in arquivo.readlines():
    print(linha)
arquivo.close()
Ln: 3 Col: 29
```

Figura 21 - Exibição dos dados lidos do arquivo com laço `for` e método `readlines()`.

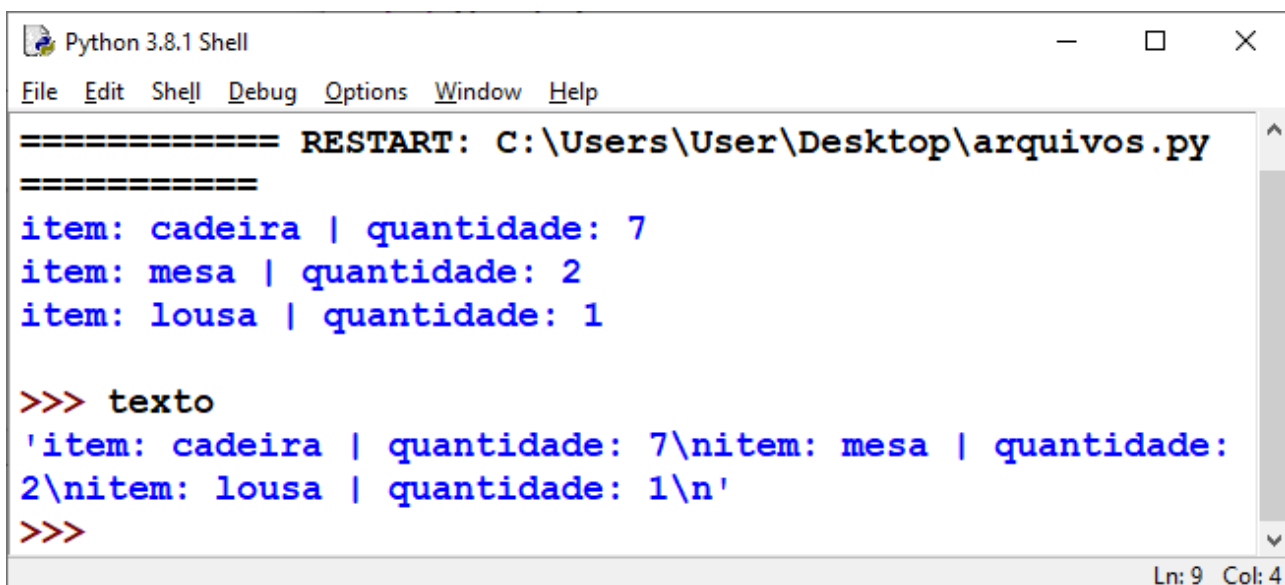
Por fim, é possível fazer a leitura do arquivo inteiro como uma única *string* que contenha todas as linhas. Para isso podemos usar o método `read()`. A Figura 22 contém o código exemplo e a Figura 23 o resultado da execução com a comprovação de que o conteúdo da variável `texto` é uma única *string*.



```
File Edit Format Run Options Window Help
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
texto = arquivo.read()
arquivo.close()
print(texto)
```

Ln: 1 Col: 55

Figura 22 - Leitura dos dados de um arquivo com o método `read()`.



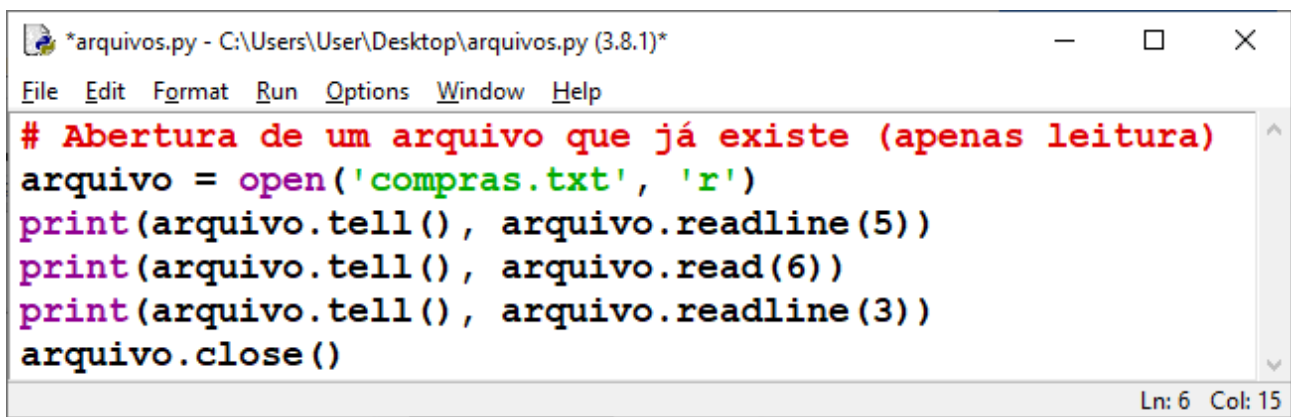
```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
===== RESTART: C:\Users\User\Desktop\arquivos.py
=====
item: cadeira | quantidade: 7
item: mesa | quantidade: 2
item: lousa | quantidade: 1

>>> texto
'item: cadeira | quantidade: 7\nitem: mesa | quantidade: 2\nitem: lousa | quantidade: 1\n'
>>>
```

Ln: 9 Col: 4

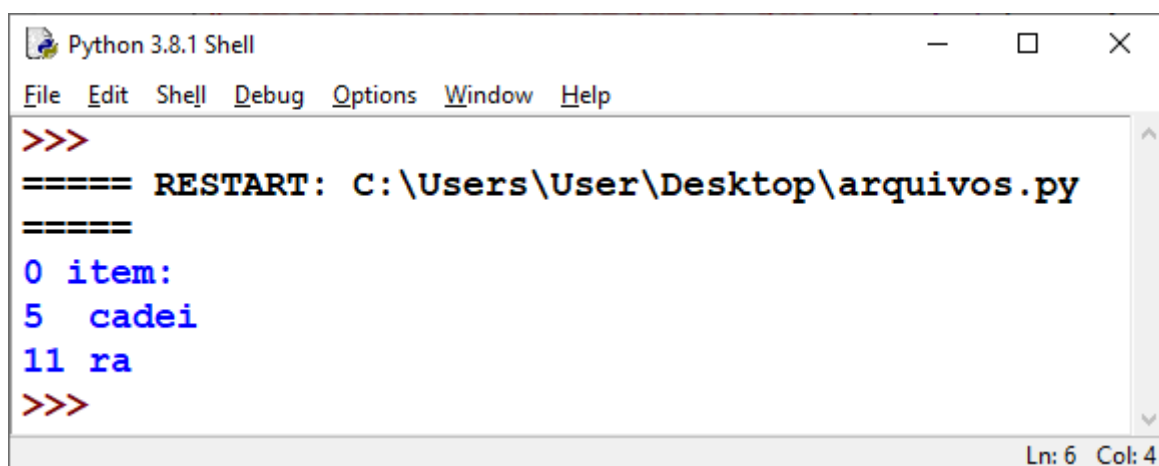
Figura 23 - Execução da codificação da Figura 22.

Os métodos `readline()` e `read()` possuem um parâmetro opcional, que indica o máximo de caracteres que deverão ser lidos ao executar o método. Por exemplo, a execução do `readline(5)` ou `read(5)` lerá apenas os primeiros cinco caracteres a partir da posição atual do marcador do arquivo, deslocando-o em cinco posições. Veja na Figura 24 e Figura 25 um exemplo de codificação e a correspondente execução.



```
# Abertura de um arquivo que já existe (apenas leitura)
arquivo = open('compras.txt', 'r')
print(arquivo.tell(), arquivo.readline(5))
print(arquivo.tell(), arquivo.read(6))
print(arquivo.tell(), arquivo.readline(3))
arquivo.close()
```

Figura 24 - Leitura dos dados de um arquivo com o métodos `readline()` e `read()` com parâmetros.



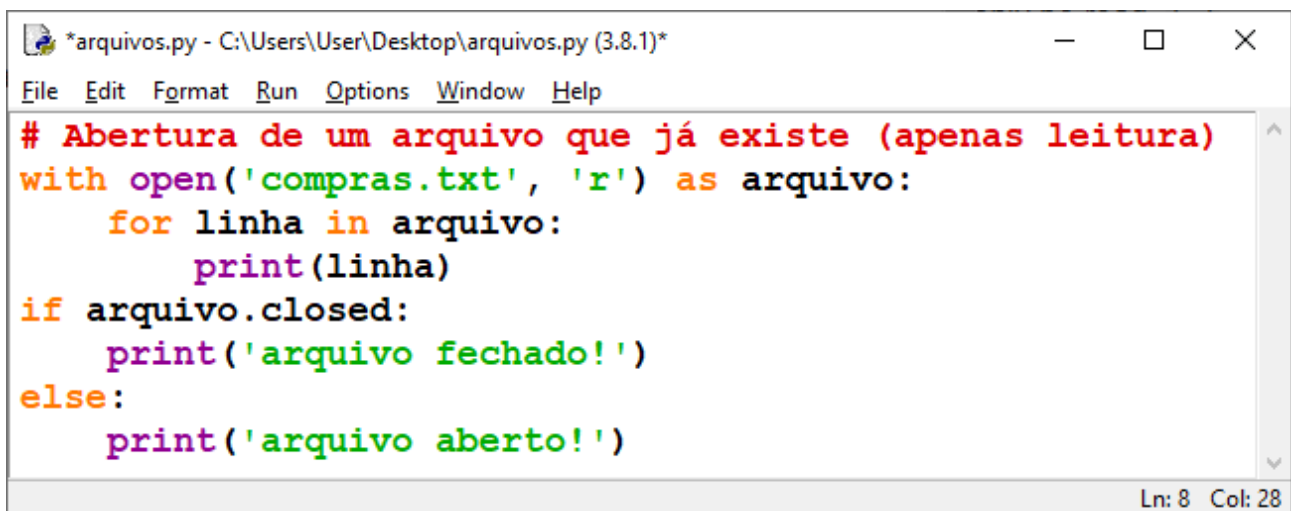
```
>>>
===== RESTART: C:\Users\User\Desktop\arquivos.py
=====
0 item:
5 cadei
11 ra
>>>
```

Figura 25 - Execução da codificação da Figura 24.

GARANTINDO O FECHAMENTO DO ARQUIVO

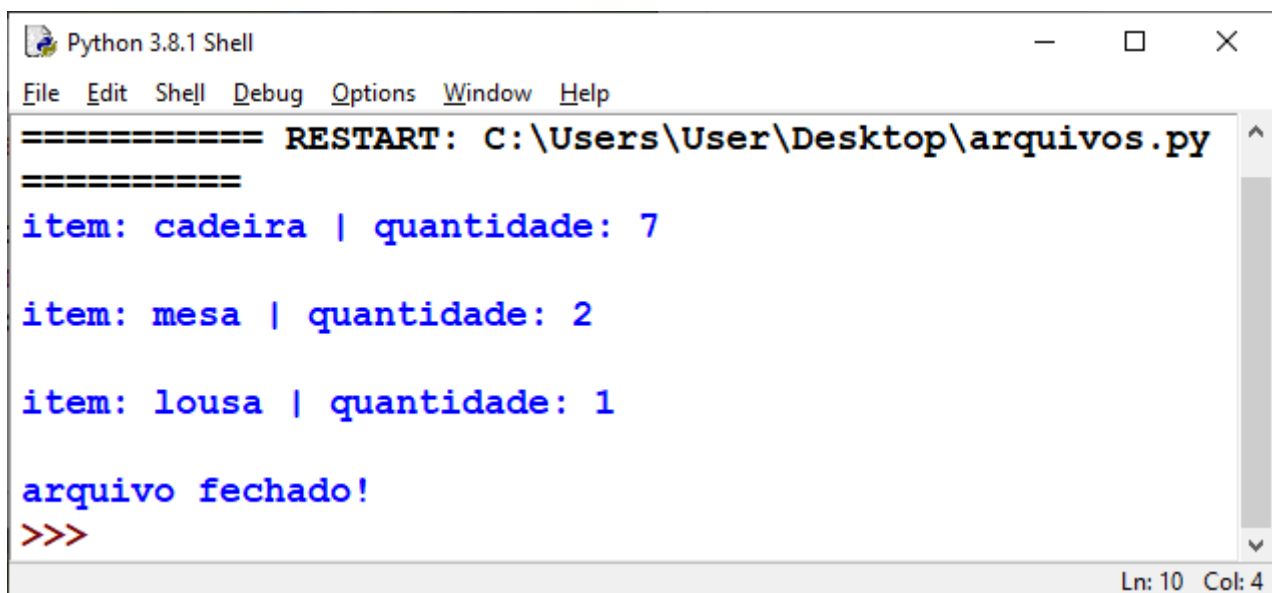
A importância em fechar um arquivo aberto já foi explicada. Porém, existe uma forma mais segura de garantir que um arquivo será fechado, mesmo que o programa encerre inesperadamente por causa de algum erro. Basta usar a palavra-chave `with`, veja na Figura 26 um exemplo de codificação e na Figura 27 a respectiva execução.

Note que não é necessário usar o método `close()`, porque após a execução da última instrução do bloco de código do comando `with` o arquivo é automaticamente fechado, mesmo que ocorra algum erro com as instruções em tempo de execução. Isso é ótimo! É uma boa prática usar `with` com arquivos.



```
# Abertura de um arquivo que já existe (apenas leitura)
with open('compras.txt', 'r') as arquivo:
    for linha in arquivo:
        print(linha)
if arquivo.closed:
    print('arquivo fechado!')
else:
    print('arquivo aberto!')
```

Figura 26 - Arquivo aberto usando a palavra-chave with.



```
===== RESTART: C:\Users\User\Desktop\arquivos.py
=====
item: cadeira | quantidade: 7

item: mesa | quantidade: 2

item: lousa | quantidade: 1

arquivo fechado!
>>>
```

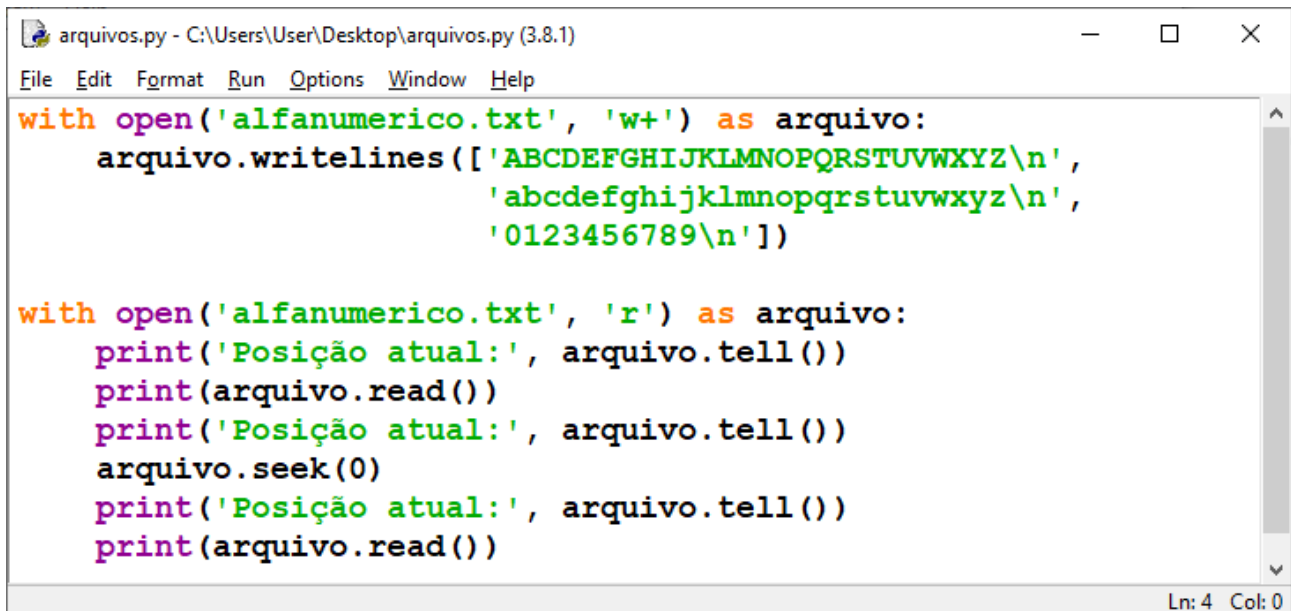
Figura 27 - Execução da codificação da Figura 26.

Novidade: O atributo `closed` devolve um valor booleano indicando se o arquivo está fechado.

ALTERANDO O MARCADOR DE POSIÇÃO ATUAL DO ARQUIVO

Como explicado anteriormente, quando uma leitura é feita no arquivo usando `readline()`, `readlines()` ou `read()`, o marcador do arquivo é alterado de modo a indicar a posição do próximo caractere que será lido. Até agora, caso fosse necessário retornar ao início do arquivo, ou seja, a posição zero, teríamos que fechá-lo e abri-lo novamente.

Porém, há uma alternativa para situações em que é preciso alterar o marcador do arquivo sem ter que fechá-lo. Basta usar o método `seek()`. O modo mais simples de usar `seek()` é passando apenas um argumento à função, que será um número natural indicando a nova posição do marcador. Logo, uma forma de retornar ao início do arquivo é usando o método `seek(0)`. Veja na Figura 28 um exemplo de codificação e na Figura 29 o exemplo executado.



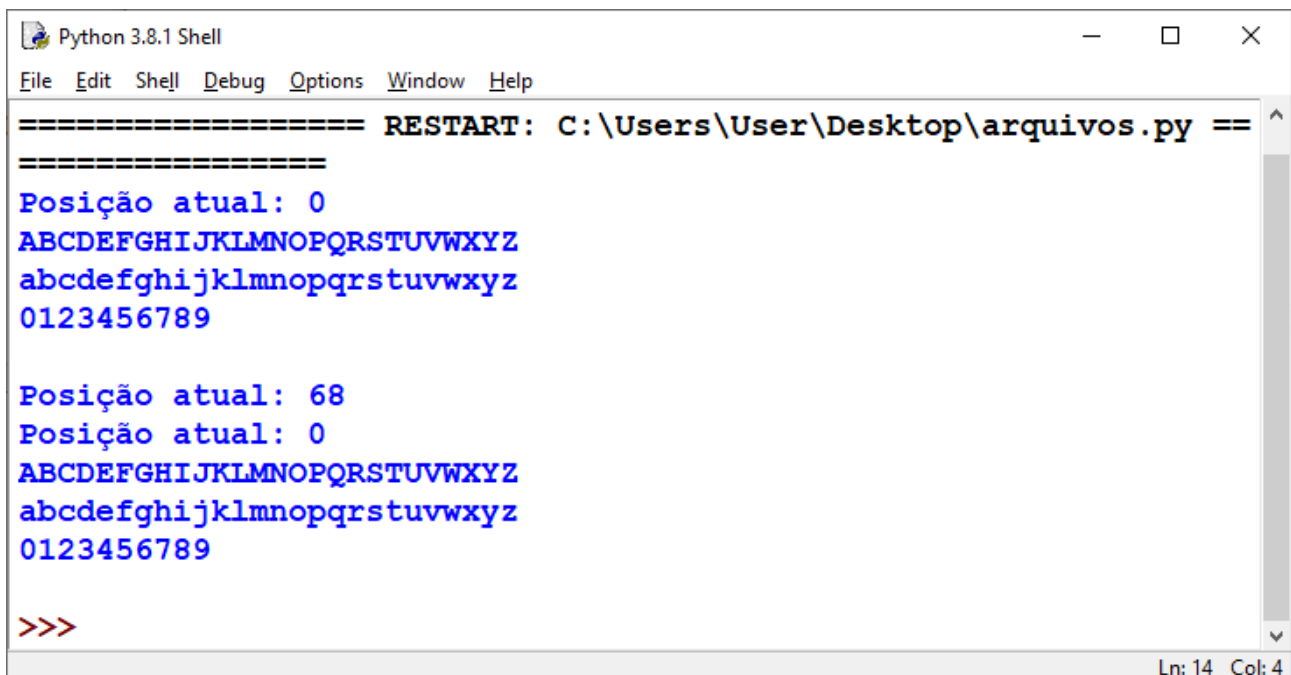
```
arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)
File Edit Format Run Options Window Help

with open('alfanumerico.txt', 'w+') as arquivo:
    arquivo.writelines(['ABCDEFGHJKLMNOPQRSTUVWXYZ\n',
                       'abcdefghijklmnopqrstuvwxyz\n',
                       '0123456789\n'])

with open('alfanumerico.txt', 'r') as arquivo:
    print('Posição atual:', arquivo.tell())
    print(arquivo.read())
    print('Posição atual:', arquivo.tell())
    arquivo.seek(0)
    print('Posição atual:', arquivo.tell())
    print(arquivo.read())

Ln: 4 Col: 0
```

Figura 28 - Leitura de um arquivo e deslocamento com o método `seek()`.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help

===== RESTART: C:\Users\User\Desktop\arquivos.py ==
=====
Posição atual: 0
ABCDEFGHJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789

Posição atual: 68
Posição atual: 0
ABCDEFGHJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789

>>>

Ln: 14 Col: 4
```

Figura 29 - Execução da codificação da Figura 28.

Novidade: O método `writelines()` é semelhante ao `write()`, porém recebe como parâmetro uma lista de *strings*, em que cada item da lista será uma linha persistida no arquivo. É útil para escrever diversas linhas sem escrever diversos `write()`.

ARQUIVOS CSV

Um tipo de arquivo de texto bastante utilizado é o CSV (*Comma Separated Values* ou Valores Separados por Vírgulas), muitas vezes gerado a partir de *softwares* de planilhas eletrônicas como o Microsoft Excel.

Uma das utilidades dos arquivos CSV é servir de meio para a transferência de dados entre diferentes programas, principalmente por ser um formato de simples manipulação por *softwares* e fácil entendimento para os humanos.

Como o próprio nome evidencia, os CSV são arquivos de texto onde cada linha possui um ou mais valores separados por vírgulas (ou outro caractere que indique a separação entre os valores). Para uma analogia com planilhas, imagine que cada linha da planilha é uma linha no arquivo CSV e cada coluna da planilha é um dos valores que compõem uma parte de determinada linha do CSV. Veja a Figura 31 que ilustra essa abstração.

	A	B	C	D
1	Ana	7.5	10.0	True
2	Bia	5.9	9.9	False
3	Clô	9.3	9.6	False
4				

Planilha

*Sem título - Bloco de Notas				
Arquivo	Editar	Formatar	Exibir	Ajuda
Ana, 7.5, 10.0, True				
Bia, 5.9, 9.9, False				
Clô, 9.3, 9.6, False				

CSV

Figura 30 - Comparação de uma planilha eletrônica com um arquivo CSV.

Cada coluna da planilha, assim como no arquivo CSV, tem um significado que geralmente está associado ao nome da coluna. Podemos supor que na Figura 30 há uma planilha para registro de

alunos de uma disciplina. Na primeira coluna podemos definir que o dado corresponde ao nome de um aluno; na segunda coluna há a nota da média dos trabalhos desse aluno; na 3ª coluna há a nota da prova e; na última coluna há um dado indicando se o aluno está reprovado por faltas.

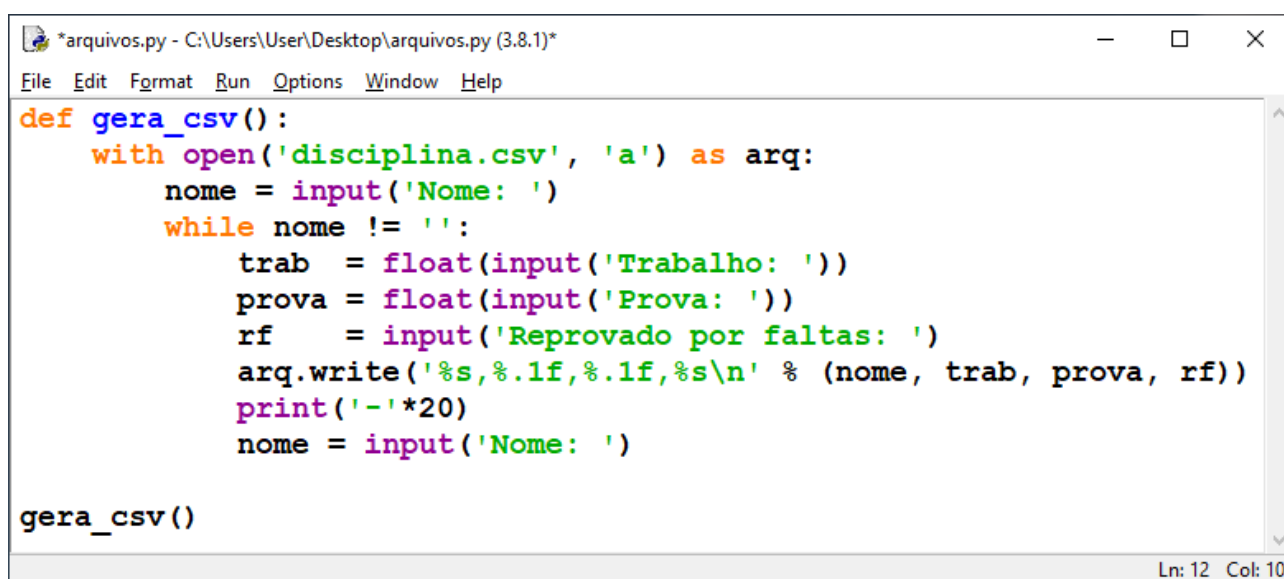
ESCREVENDO UM ARQUIVO EM FORMATO CSV

Vamos supor que queremos gravar um arquivo CSV com alguns dados de alunos de uma disciplina. Os campos que compõem o arquivo são descritos na Tabela 2.

NOME DO DADO	TIPO DO DADO
Nome	Cadeia de caracteres
Média dos trabalhos	Número real
Nota da prova	Número real
Reprovado por falta	Valor booleano

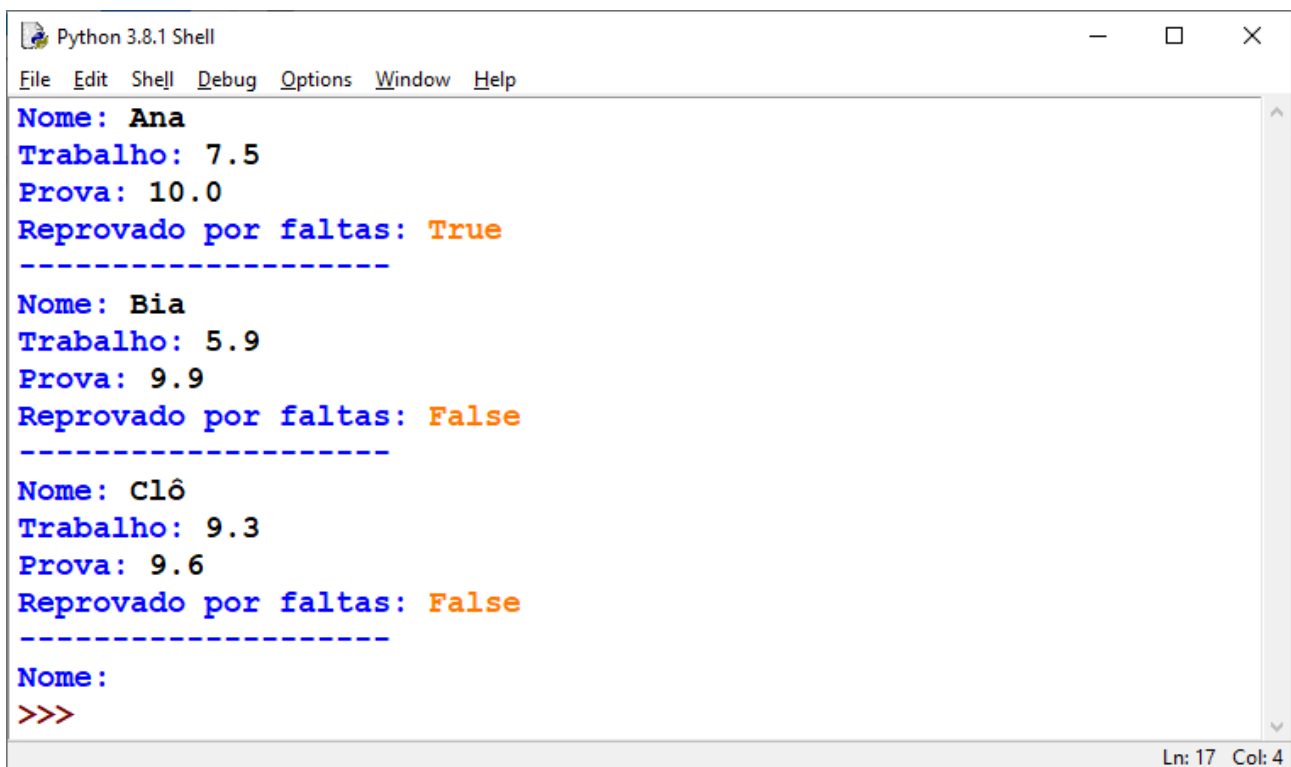
Tabela 2 - Dados que comporão o arquivo CSV.

Uma proposta de solução para gerar esse arquivo está na Figura 31, onde dados são inseridos até que uma *string* vazia seja atribuída à variável `nome`. Note que a abertura do arquivo foi feita com `'a'`, portanto, caso já existam linhas no arquivo, as novas serão anexadas ao final. A figura 32 ilustra a execução do programa proposto e a Figura 33 o arquivo gerado.



```
*arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)*
File Edit Format Run Options Window Help
def gera_csv():
    with open('disciplina.csv', 'a') as arq:
        nome = input('Nome: ')
        while nome != '':
            trab = float(input('Trabalho: '))
            prova = float(input('Prova: '))
            rf = input('Reprovado por faltas: ')
            arq.write('%s,%.1f,%.1f,%s\n' % (nome, trab, prova, rf))
            print('-'*20)
            nome = input('Nome: ')
gera_csv()
```

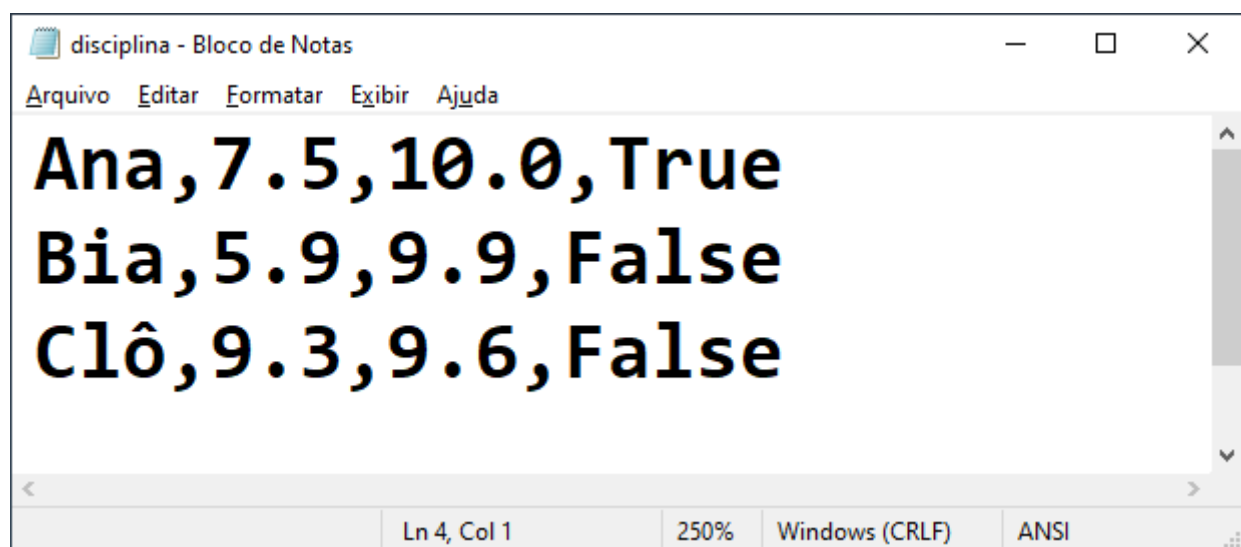
Figura 31 - Criação de um arquivo CSV e escrita de dados inseridos pelo usuário.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Nome: Ana
Trabalho: 7.5
Prova: 10.0
Reprovado por faltas: True
-----
Nome: Bia
Trabalho: 5.9
Prova: 9.9
Reprovado por faltas: False
-----
Nome: Clô
Trabalho: 9.3
Prova: 9.6
Reprovado por faltas: False
-----
Nome:
>>>
```

Ln: 17 Col: 4

Figura 32 - Execução da codificação da Figura 31.



```
disciplina - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
Ana,7.5,10.0,True
Bia,5.9,9.9,False
Clô,9.3,9.6,False
```

Ln 4, Col 1 250% Windows (CRLF) ANSI

Figura 33 - Arquivo gerado com a codificação da Figura 31.

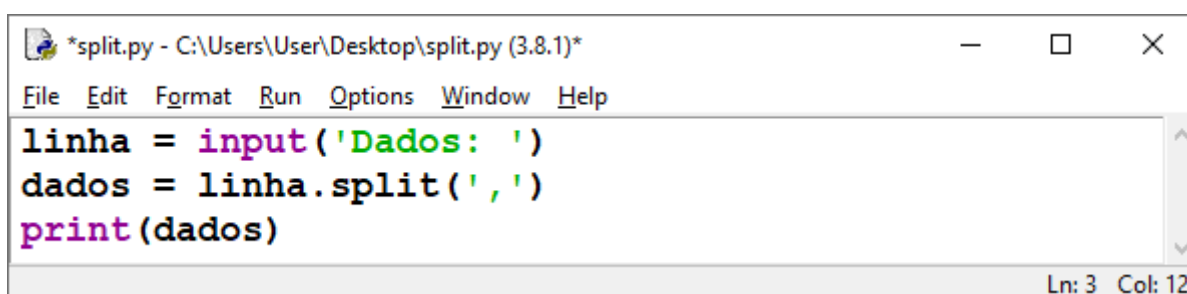
LENDO UM ARQUIVO EM FORMATO CSV

Agora, nossa intenção é ler os dados do arquivo CSV gerado pelo programa anterior e executar operações com esses dados. Para isso, será necessário abrir o arquivo em modo leitura e dividir toda linha lida em "colunas", para que cada dado seja convertido para o tipo certo e guardado em

uma variável. Para tanto, precisamos primeiro aprender como dividir os dados de uma *string* de acordo com um separador definido.

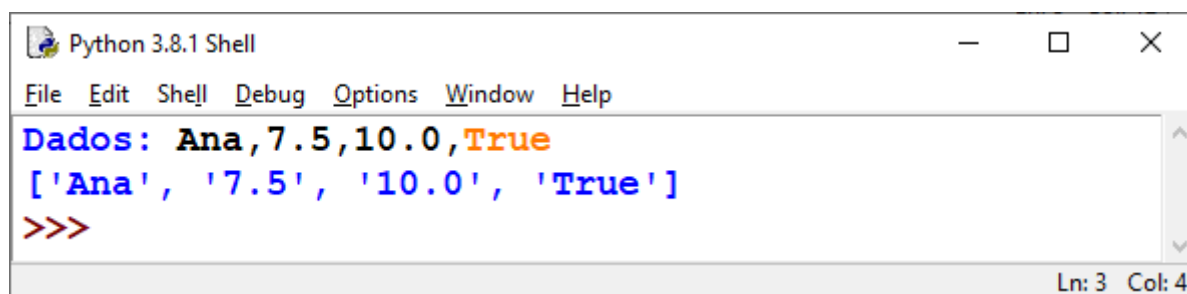
- **Dividindo uma *string***

Para dividir uma *string* que contém separadores, usamos o método `split()`, que recebe como argumento outra *string* que define o separador de valores. O método `split()` devolve como resposta uma lista preenchida com os valores da *string* que foi dividida. Veja na Figura 34 um exemplo de codificação e na Figura 35 a correspondente execução.



```
*split.py - C:\Users\User\Desktop\split.py (3.8.1)*
File Edit Format Run Options Window Help
linha = input('Dados: ')
dados = linha.split(',')
print(dados)
Ln: 3 Col: 12
```

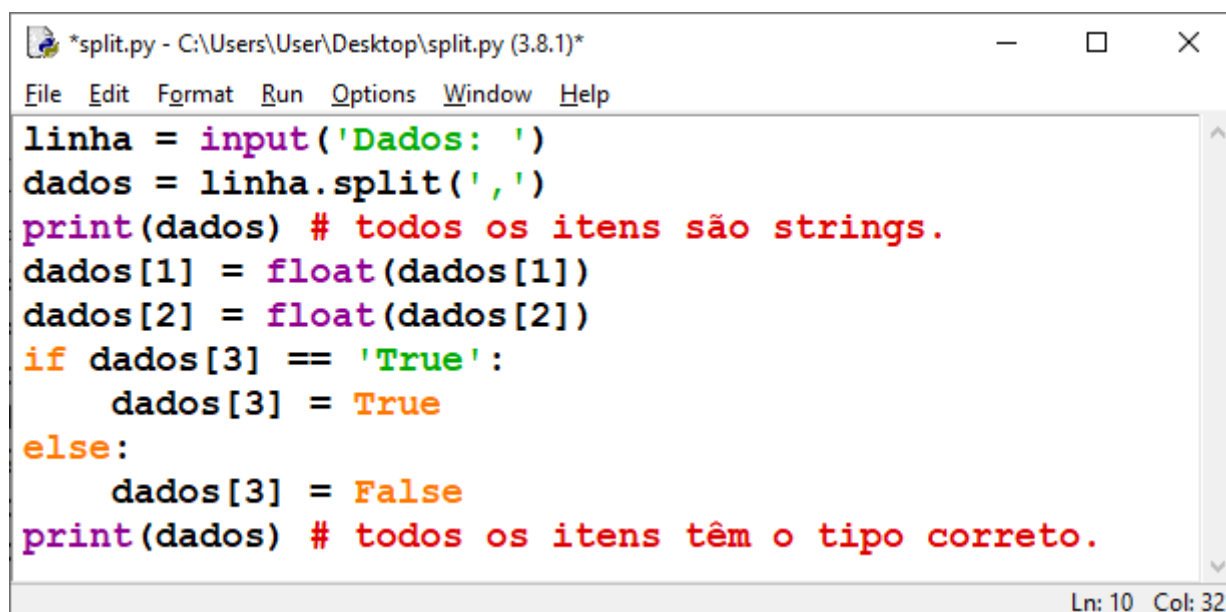
Figura 34 - Dividindo uma *string* com base nos separadores usando `split()`.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Dados: Ana,7.5,10.0,True
['Ana', '7.5', '10.0', 'True']
>>>
Ln: 3 Col: 4
```

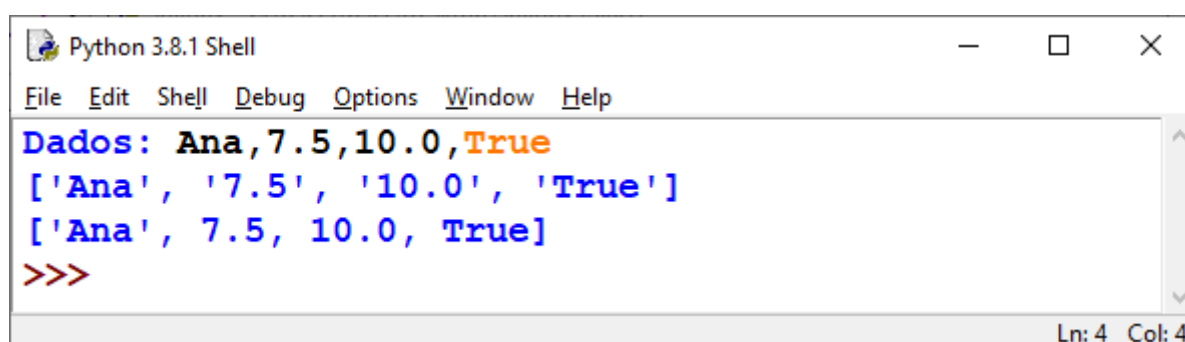
Figura 35 - Execução da codificação da Figura 34.

Indicamos `,` como separador ao `split()`, porém poderíamos escolher qualquer outro, desde que correspondesse corretamente ao conteúdo do arquivo. Note que todos os valores originados da divisão da *string* `linha` e armazenados na lista `dados` continuam sendo do tipo *string*, portanto, caso seja necessário executar operações sobre os valores, deve-se convertê-los para os tipos desejados. Veja na Figura 36 e Figura 37 um exemplo de como proceder com a conversão atualizando a própria lista `dados`.



```
*split.py - C:\Users\User\Desktop\split.py (3.8.1)*
File Edit Format Run Options Window Help
linha = input('Dados: ')
dados = linha.split(',')
print(dados) # todos os itens são strings.
dados[1] = float(dados[1])
dados[2] = float(dados[2])
if dados[3] == 'True':
    dados[3] = True
else:
    dados[3] = False
print(dados) # todos os itens têm o tipo correto.
Ln: 10 Col: 32
```

Figura 36 - Dividindo os dados da linha com `split()` e convertendo para o tipo correto.

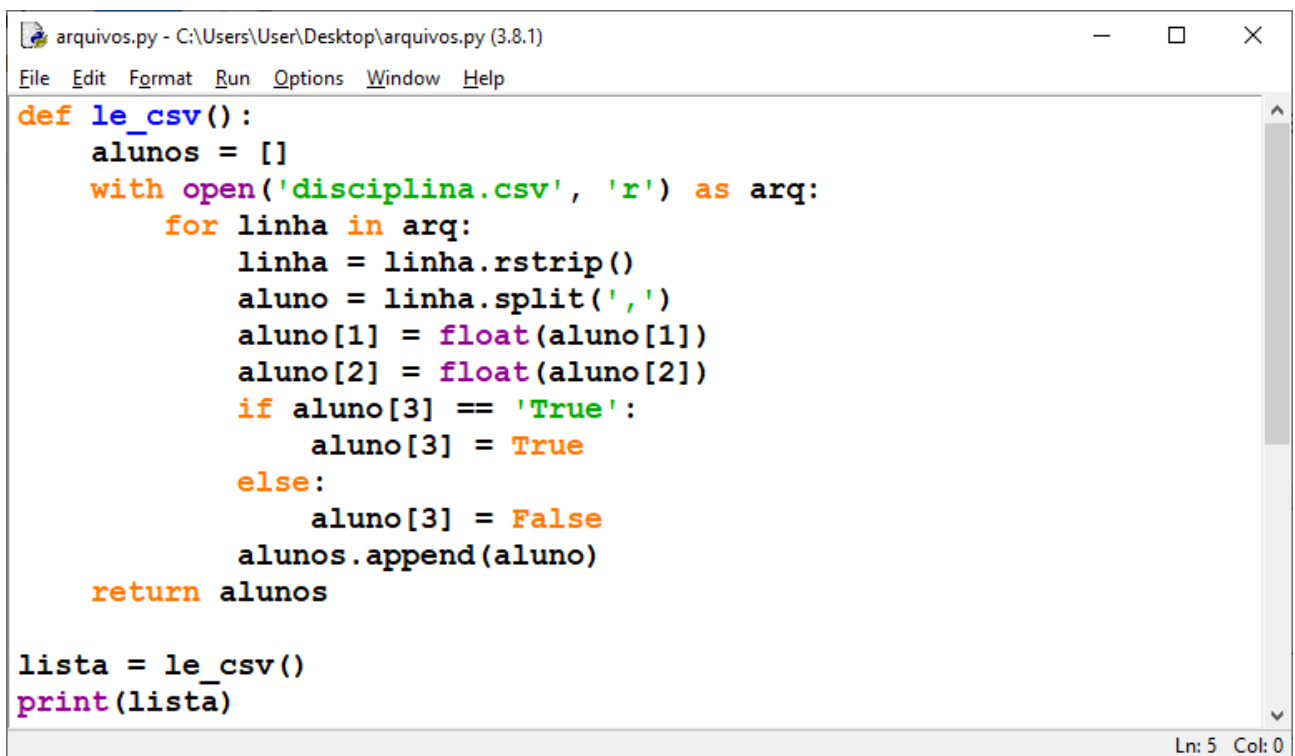


```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Dados: Ana,7.5,10.0,True
['Ana', '7.5', '10.0', 'True']
['Ana', 7.5, 10.0, True]
>>>
Ln: 4 Col: 4
```

Figura 37 - Execução da codificação da Figura 36.

Obs.: É claro que os itens da lista que devem ser *strings* não precisam ser convertidos, por isso não houve conversão de `dados[0]`.

Agora podemos retomar nosso arquivo `disciplina.csv` e executar operações com seus dados. Porém, ainda há um problema: como retirar o `'\n'` do final de cada linha lida do arquivo? Pois esse caractere ficará acoplado ao último item da lista gerada por `split()`. Simples! Basta usar o método `rstrip()` que retira espaços em branco e quebras de linhas à direita de uma *string*. Veja a codificação do programa que faz a leitura e sua execução na Figura 38 e Figura 39, respectivamente.

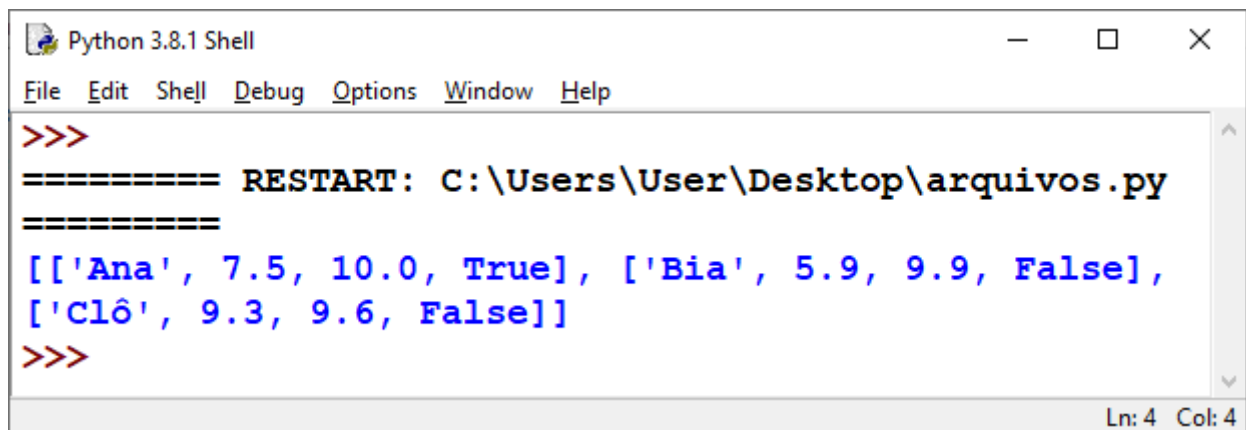


```
def le_csv():
    alunos = []
    with open('disciplina.csv', 'r') as arq:
        for linha in arq:
            linha = linha.rstrip()
            aluno = linha.split(',')
            aluno[1] = float(aluno[1])
            aluno[2] = float(aluno[2])
            if aluno[3] == 'True':
                aluno[3] = True
            else:
                aluno[3] = False
            alunos.append(aluno)
    return alunos

lista = le_csv()
print(lista)
```

Ln: 5 Col: 0

Figura 38 - Leitura do arquivo CSV com valores armazenados com o tipo correto em uma lista.



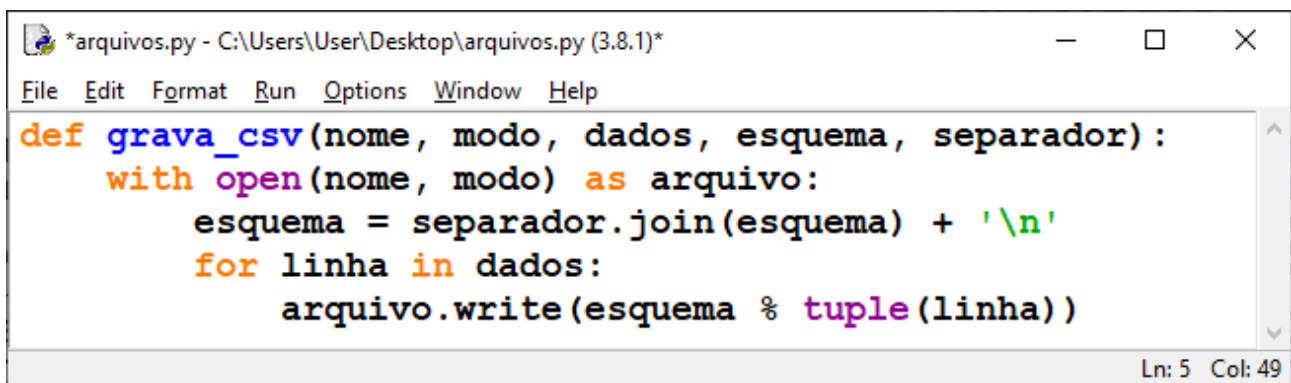
```
>>>
===== RESTART: C:\Users\User\Desktop\arquivos.py
=====
[['Ana', 7.5, 10.0, True], ['Bia', 5.9, 9.9, False],
 ['Clô', 9.3, 9.6, False]]
>>>
```

Ln: 4 Col: 4

Figura 39 - Execução da codificação da Figura 38.

Note que a função `le_csv()` devolve uma lista em que cada item corresponde à uma linha do arquivo CSV. Também é importante perceber que os valores de cada linha foram convertidos para o tipo correto e para isso foi necessário saber a **esquematização** do arquivo, ou seja, como ele estava organizado.

Agora, os dados podem ser manipulados e persistidos em um arquivo CSV. A Figura 40 contém uma função de gravação baseada naquela da Figura 31, porém mais flexível e concisa, pois recebe como parâmetros o nome do CSV que será gerado (*string*), o modo de abertura (*string*), os dados que serão gravados (lista de listas), o **esquema** do arquivo (lista de *strings*) e o separador (*string*).

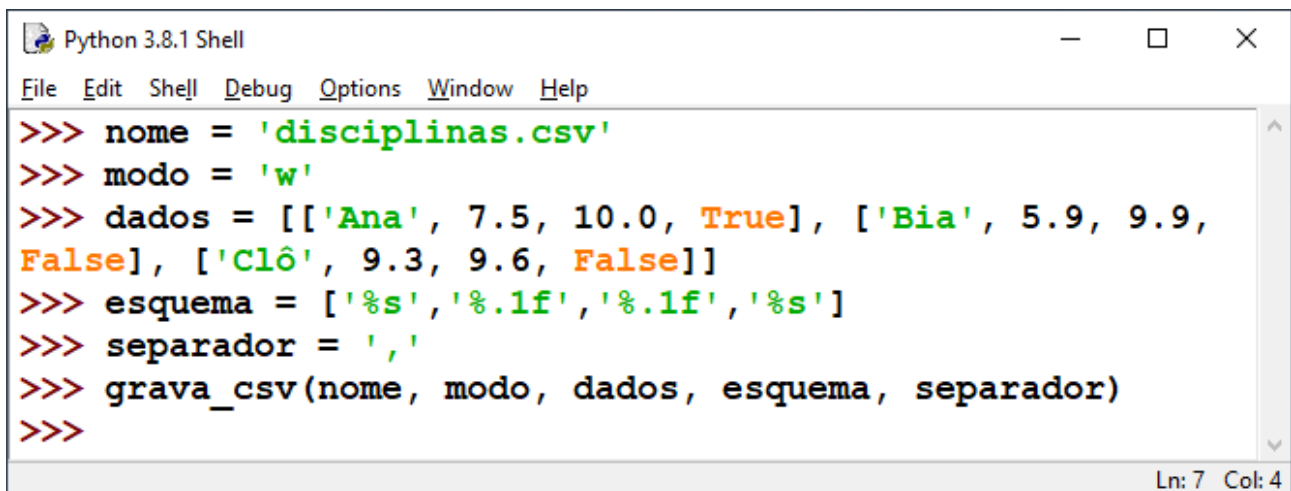


```
*arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)*
File Edit Format Run Options Window Help
def grava_csv(nome, modo, dados, esquema, separador):
    with open(nome, modo) as arquivo:
        esquema = separador.join(esquema) + '\n'
        for linha in dados:
            arquivo.write(esquema % tuple(linha))
Ln: 5 Col: 49
```

Figura 40 - Função para gravação dos dados em um arquivo CSV.

Novidade: O método `join()` recebe uma sequência como argumento e devolve como resposta uma *string* composta pelos itens do argumento unidos pelo separador em que o método é aplicado.

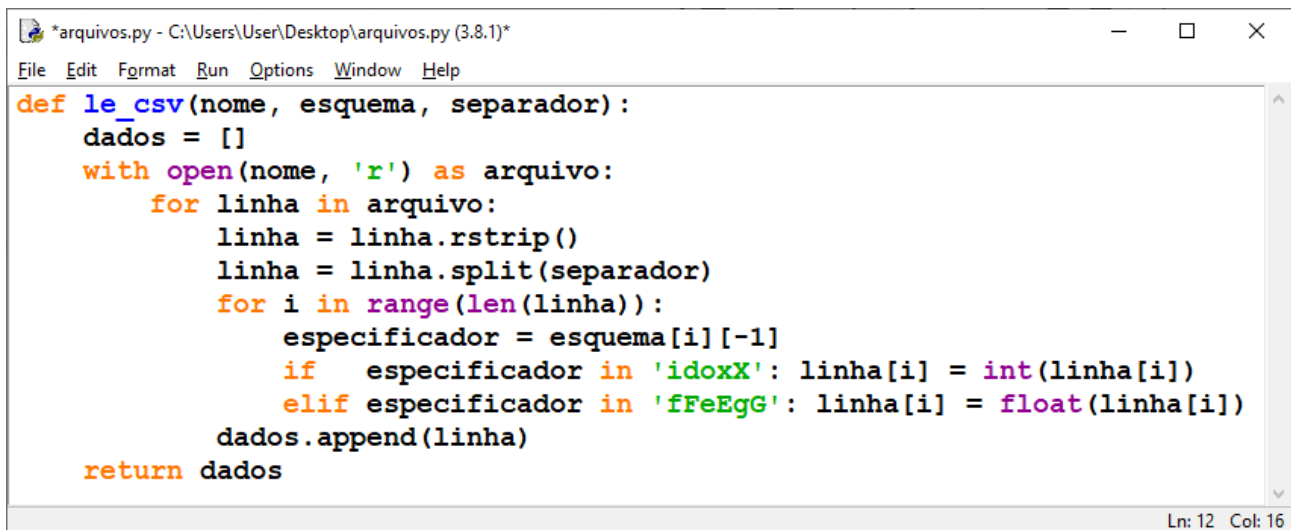
Na Figura 41 há um exemplo de utilização da função ilustrada na Figura 40. Note, como o esquema do CSV é personalizável, por meio do parâmetro de mesmo nome, podemos montar o arquivo com a quantidade de itens que quisermos e com formatações diversas, bastando construir uma lista com os especificadores adequados. Além disso, o separador também é personalizável, não ficando restrito somente às vírgulas.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
>>> nome = 'disciplinas.csv'
>>> modo = 'w'
>>> dados = [['Ana', 7.5, 10.0, True], ['Bia', 5.9, 9.9, False], ['Clô', 9.3, 9.6, False]]
>>> esquema = ['%s', '%.1f', '%.1f', '%s']
>>> separador = ','
>>> grava_csv(nome, modo, dados, esquema, separador)
>>>
Ln: 7 Col: 4
```

Figura 41 - Teste da função ilustrada na Figura 40.

Uma forma simples e flexível de abrir um arquivo CSV com qualquer padronização de separador é ilustrada na Figura 42 com o respectivo teste de execução na Figura 43.

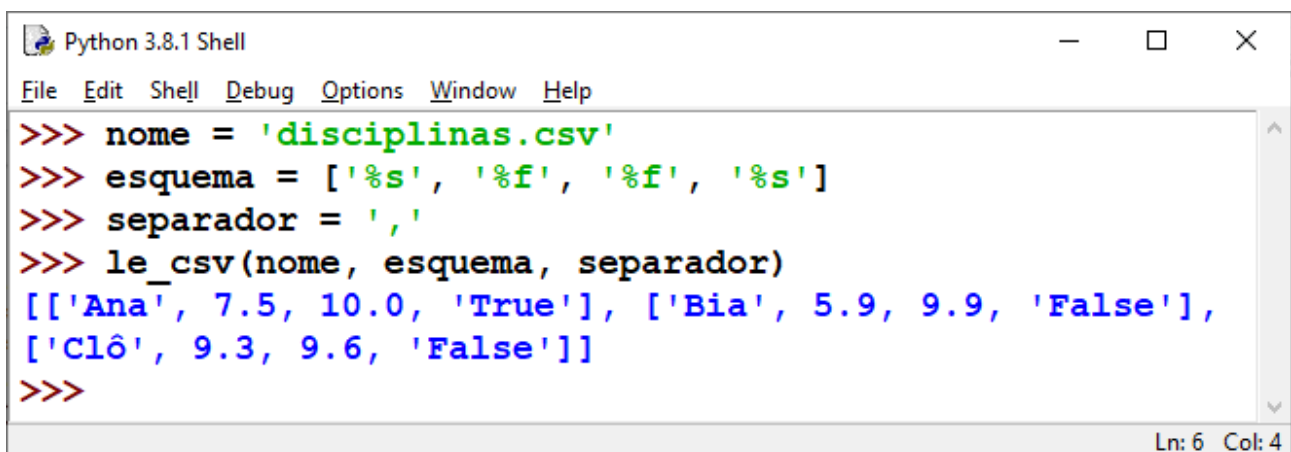


```
*arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)*
File Edit Format Run Options Window Help

def le_csv(nome, esquema, separador):
    dados = []
    with open(nome, 'r') as arquivo:
        for linha in arquivo:
            linha = linha.rstrip()
            linha = linha.split(separador)
            for i in range(len(linha)):
                especificador = esquema[i][-1]
                if especificador in 'idoxX': linha[i] = int(linha[i])
                elif especificador in 'fFeEgG': linha[i] = float(linha[i])
            dados.append(linha)
    return dados
```

Ln: 12 Col: 16

Figura 42 - Função para a leitura dos dados de um arquivo CSV.



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help

>>> nome = 'disciplinas.csv'
>>> esquema = ['%s', '%f', '%f', '%s']
>>> separador = ','
>>> le_csv(nome, esquema, separador)
[['Ana', 7.5, 10.0, 'True'], ['Bia', 5.9, 9.9, 'False'],
 ['Clô', 9.3, 9.6, 'False']]
>>>
```

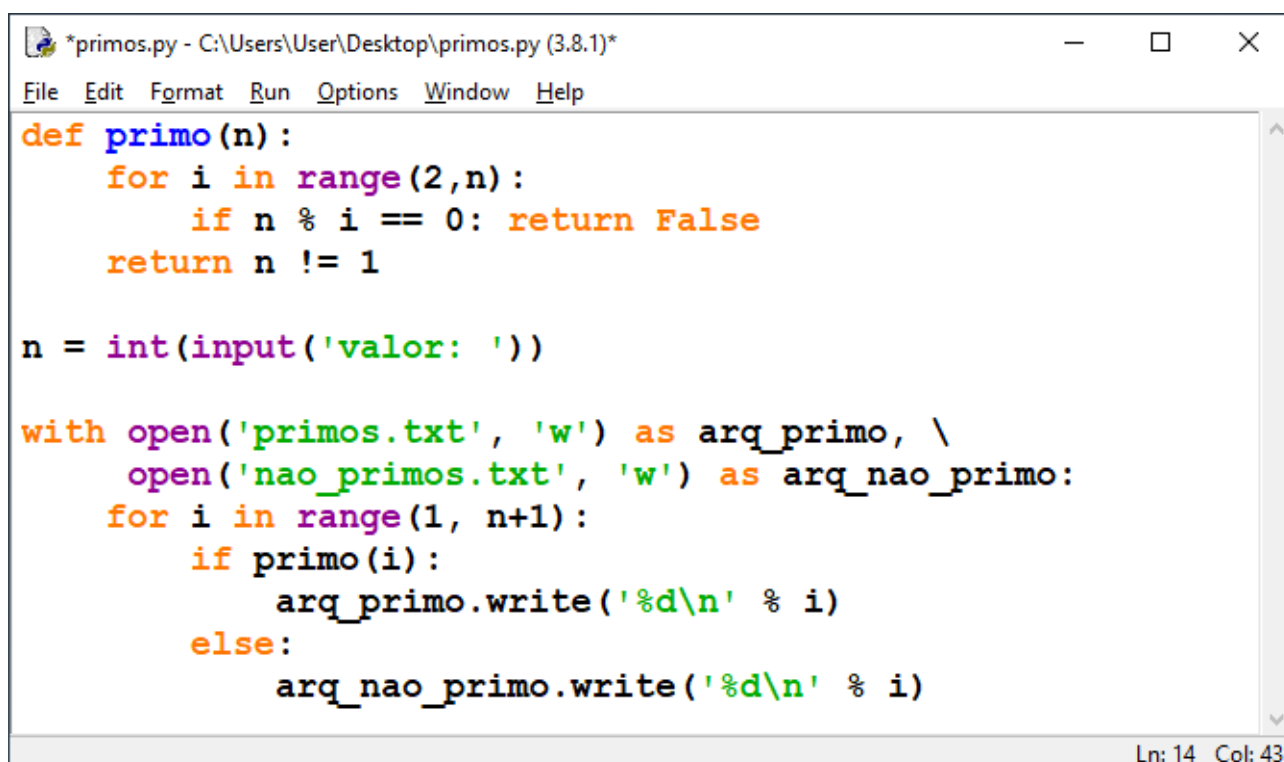
Ln: 6 Col: 4

Figura 43 - Teste da função ilustrada na Figura 42.

EXERCÍCIOS RESOLVIDOS

Os exercícios desta seção estão resolvidos, portanto teste as codificações e procure entender o funcionamento. **Conselho:** tente resolver antes de ver as soluções.

1. Crie um programa que leia um número natural n dado pelo usuário e crie dois arquivos de texto simples, o primeiro terá os números primos do intervalo $[1..n]$ e o segundo os números não primos do intervalo $[1..n]$. A resolução está ilustrada na Figura 44.



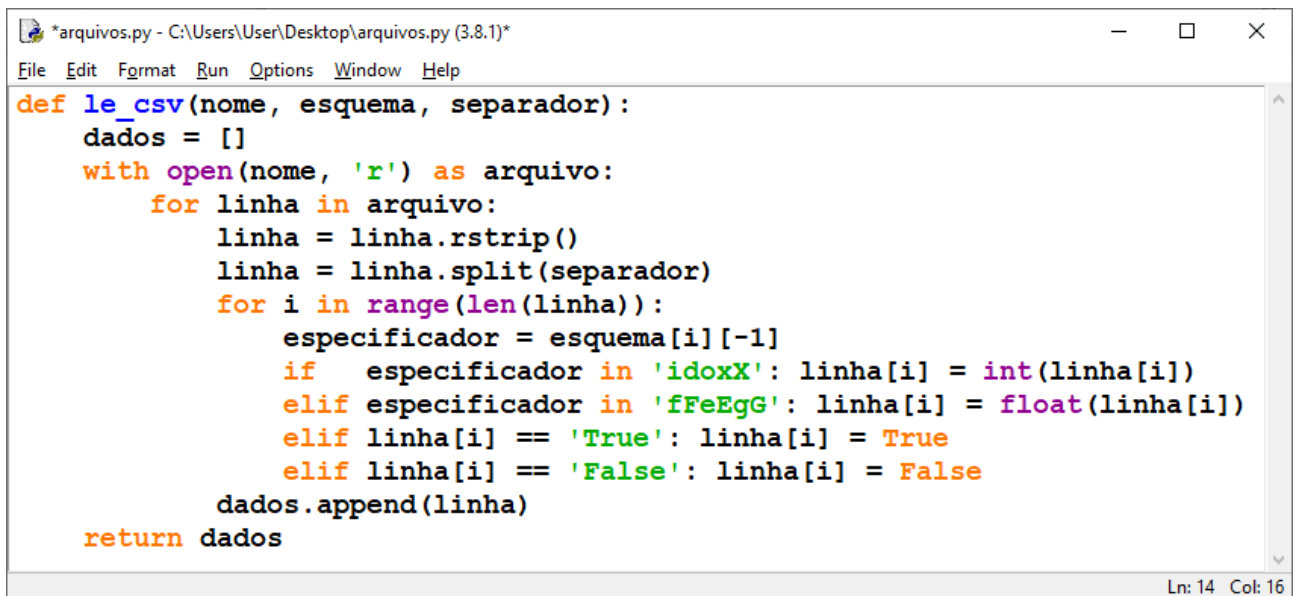
```
*primos.py - C:\Users\User\Desktop\primos.py (3.8.1)*
File Edit Format Run Options Window Help
def primo(n):
    for i in range(2,n):
        if n % i == 0: return False
    return n != 1

n = int(input('valor: '))

with open('primos.txt', 'w') as arq_primo, \
    open('nao_primos.txt', 'w') as arq_nao_primo:
    for i in range(1, n+1):
        if primo(i):
            arq_primo.write('%d\n' % i)
        else:
            arq_nao_primo.write('%d\n' % i)
Ln: 14 Col: 43
```

Figura 44 - resolução do Exercício 1.

2. Você deve ter notado que na codificação ilustrada na Figura 42 há um problema quando a linha do arquivo CSV possui um valor booleano entre os itens, pois não temos um especificador de formato dedicado a esse tipo em Python e, por isso, optamos por usar um de *string*. Faça uma codificação que aperfeiçoe essa função, transformando valores booleanos (True e False) realmente em valores lógicos e não *strings*. A resolução está ilustrada na Figura 45.



The image shows a screenshot of a Python IDE window titled "*arquivos.py - C:\Users\User\Desktop\arquivos.py (3.8.1)*". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The main editor area contains the following Python code:

```
def le_csv(nome, esquema, separador):  
    dados = []  
    with open(nome, 'r') as arquivo:  
        for linha in arquivo:  
            linha = linha.rstrip()  
            linha = linha.split(separador)  
            for i in range(len(linha)):  
                especificador = esquema[i][-1]  
                if especificador in 'idoxX': linha[i] = int(linha[i])  
                elif especificador in 'fFeEgG': linha[i] = float(linha[i])  
                elif linha[i] == 'True': linha[i] = True  
                elif linha[i] == 'False': linha[i] = False  
            dados.append(linha)  
    return dados
```

The status bar at the bottom right indicates "Ln: 14 Col: 16".

Figura 45 - resolução do Exercício 2.