# Python: Arquivos e pastas

Posted on 24/03/2021 by Guilherme


Todos os tópicos →

Quase sempre um programa de computador precisa recuperar dados ou informações previamente existentes, processá-los de algum modo e retornar um resultado que deve ficar armazenado para uso futuro. Esses dados podem ser obtidos de várias formas diferentes: lidos em papel impresso (através de um softaware de OCR), digitados pelo usuário, coletados ne internet ou captados por algum instrumento de medição, entre outras. O resultado final também pode ser impresso ou exibido na tela. Mas, como maior frequência, a informação inicial e final é lida e gravada em disco, em arquivos de texto ou binários, planilhas e bancos de dados (entre outros muitos formatos).

Nesse artigo, e em todos os demais no site, usamos as palavras *pasta* e *diretório* como sinônimos. A expressão *diretório* (*directory*) é mais antiga e foi usada no Unix, no Linux e no Windows/DOS, que mais tarde criou o termo *pasta* (ou *folder*).

## O objeto arquivo (*file*)

# Abrir e ler um arquivo

No Python existem diversas funções no módulo básico para abrir, manipular e escrever em arquivos. Um arquivo aberto é um objeto que pode ser atribuído à uma variável e, por meio dessa variável, as propriedades e métodos do objeto podem ser acessadas. A função `open()` estabelece uma conexão com o arquivo no disco.

```
obj = open('caminho_para_arquivo', modo)
```

Os modos são listados abaixo. Suponha, por exemplo, que existe na pasta de trabalho ativa um arquivo com nome `alunos.txt` contendo informações sobre alunos de um curso. Seu conteúdo pode ser exibido da seguinte forma:

```
» objeto_arquivo = open('alunos.txt', 'r')
» texto = objeto_arquivo.read()
» objeto_arquivo.close()
» print(texto)

↳ id, nome     , cidade         , idade , nota
↳ 10, Pedro    , São Paulo      , 34    , 83.0
↳ 11, Maria    , São Paulo      , 23    , 59.0
↳ 12, Janaina  , Rio de Janeiro , 32    , 86.0
↳ 13, Wong     , Brasília       , 43    , 89.0
↳ 14, Roberto  , Salvador       , 38    , 98.0
↳ 15, Marco    , Curitiba       , 31    , 61.0
↳ 16, Paula    , Belo Horizonte , 34    , 44.0
```

**O que é um arquivo no Python?** Um arquivo é um conjunto de informações ou dados gravados no dispositivos de armazenamento do computador, como o disco rígido ou um *ssd*. Ele também pode estar armazenado em memória *rom*, pronto para a manipulação. O nome do arquivo é uma referência a uma tabela que informa onde estão estes dados. A função interna do Python `open()` é usada para abrir um arquivo. Ela retorna um objeto arquivo, também chamado de identificador ou *handle*, usado para ler ou modificar o arquivo. O método `read()` lê o arquivo e retorna para a variável `texto`.

Após o uso, o arquivo deve ser fechado com o método `close()`. O fechamento do objeto libera recursos da máquina e permite que o arquivo seja acessado por outro bloco de código. Em alguns casos pode ocorrer que alterações feitas

pelo código não sejam gravadas em disco até que o arquivo seja fechado. Além disso, um problema técnico, como a falta de energia, pode ocasionar em perda de dados.

Estritamente dizendo o mais adequado para lidar com arquivos seria um bloco de código do seguinte tipo.

```
» try:
»     f = open("data.txt", "w")
»     # código com as operações necessárias usando o arquivo f
» finally:
»     f.close()
```

Isso evita a possibilidade de que algum erro faça com que o comando de fechamento seja pulado. Uma forma mais compacta de escrever isso, e que evita o problema citado, usa o comando `with` . No Python a declaração `with` é usada para manipular exceções e tornar o código mais enxuto e de fácil leitura.

```
# exemplo de código usando with
» with open('data.txt', 'r') as f:
»     <codigo usando o arquivo f>
»     # (pronto!)
```

O fechamento nesse caso é automático. O comando `with` facilita a escrita do código que envolve recursos que devem ser finalizados. Essa finalização significa liberação de recursos de memória e evita que erros desnecessários sejam introduzidos.

## Ler linhas de um arquivo

Também é possível ler o arquivo com `readlines()` que retorna uma lista de linhas que podem ser percorridas depois.

```
# todas as linhas do arquivo são impressas dessa forma
» with open('alunos.txt', 'r') as f:
»     linhas = f.readlines()
» for linha in linhas:
»     print(linha, end ='')
# as linhas acima são impressas (mas foram aqui omitidas)

# como readlines() retorna uma lista de linhas ele pode
# ser usado para contar quantas linhas existem no arquivo
```

```
» with open('alunos.txt', 'r') as f:
»     print(len(f.readlines()))
↳ 8
```

Uma forma alternativa consiste em ler o arquivo uma linha de cada vez, processando as linhas na medida em que são lidas.

```
» with open('alunos.txt', 'r') as f:
»     print('1-', f.readline(), end='')
»     print('2-', f.readline(), end='')

↳ 1- id, nome     , cidade         , idade , nota
↳ 2- 10, Pedro    , São Paulo      , 34    , 83.0
```

Uma linha é o texto que termina a cada um sinal de quebra linha.

> Quebras de linhas (end of line, *EOL*) são caracteres especiais inseridos em arquivos para indicar o final de uma linha e o início de outra. Esses caracteres, em geral, não são vistos nos editores de texto mas marcam os finais de cada linha. Alimentação de linha (line feed, LF) e retorno de carro (carriage return, CR) são usados, dependendo do sistema operacional. LF é representado por `\n`, `0x0A` em hexadecimal ou `10` decimal. CR é representado por `\r`, `0x0D` em hexadecimal ou `13` decimal. O Linux (e outros sistemas derivados do UNIX) usa `\n`. O Windows usa `\r\n` e o OSX usa `\r`.

A cada uso do método `.readline()` uma nova linha é recuperada, até o fim do arquivo. A operação pode ser repetida até que uma string nula seja retornada. Ao final o iterador fica vazio e deve ser lido novamente caso outra iteração seja necessária. Com essa forma de leitura as linhas são lidas uma de cada vez e a requisição de memória é menor.

```
» with open('alunos.txt', 'r') as f:
»     while True:
»         linha = f.readline()
»         if not linha:
»             break
»         print(linha, end='')

# Observe que readline() é o método default do objeto arquivo.
# Isso significa que o mesmo resultado pode ser obtido com as linhas
```

```
» with open('alunos.txt', 'r') as f:
»     for linha in f:
»         print(linha, end='')
```


Em ambos os casos o arquivo dos alunos é exibido. O teste de final de arquivo pode ser excluído porque o iterador se esgota ao final e o laço `for` é abandonado.

Observe que a linha `if not linha:` tem o mesmo efeito que `if linha == '':`. Ao final da iteração uma string vazia é retornada e strings vazias são avaliadas como `False` em testes booleanos.

# Gravar em um arquivo

No código abaixo um arquivo é aberto para gravação.

```
» texto = (
»     'Texto a ser gravado em disco\n'
»     'pode conter quantas linhas se desejar\n'
»     'novas linhas são inseridas com \'newline\''
» )
» arquivo = open('teste_gravar.txt', 'w')
» arquivo.write(texto)
» arquivo.close()

# a mesma operação usando with
» with open('teste_gravar.txt', 'w') as arquivo:
»     arquivo.write(texto)
```

Após a execução dessas linhas um arquivo `teste_gravar.txt` será encontrado na pasta de trabalho. No texto a ser inserido novas linhas são criadas após o sinal de *newline* (**\n**).

# Gravação incremental em um arquivo

Para acrescentar linhas ao arquivo, sem apagar as já existentes, usamos o parâmetro `'a'` (de append):

```
# para acrescentar um novo aluno ao arquivo alunos.txt:
» novo_aluno = '17, Ana      , Belo Horizonte , 21    , 78.5\n'
» with open('alunos.txt', 'a') as f:
»     f.write(novo_aluno)
# a nova aluna, Ana e seus dados, fica acrescentada ao final do texto.
```

Observe que `write()` não insere quebras de linha automaticamente. Por isso terminamos a linha com o sinal de *nova linha* `\n` . Dessa forma novo texto inserido posteriormente já se inicia em linha própria.

Outra forma de iterar sobre as linhas de um arquivo consiste em tratar o objeto de arquivo o como um iterador em um laço `for` . O ex. mostra que `readlines()` retorna um iterável que pode ser percorrido dentro de laço for:

```
» with open('alunos.txt', 'r') as f:
»     for linha in f.readlines():
»         print(linha)
# o arquivo inteiro é exibido (mas omitido aqui)
```

Todo o texto lido em um arquivo dessa forma é uma string, mesmo que composto de dígitos. Se inteiros e números de ponto flutuante estão no texto e precisam ser usados como números, por ex. para um cálculo, eles devem ser convertidos usando-se as conversões `int()` e `float()` , respectivamente.

Um exemplo disso segue abaixo. Lembrando que o arquivo `alunos.txt` tem o conteúdo:

```
id, nome     , cidade         , idade , nota
10, Pedro    , São Paulo      , 34    , 83.0
11, Maria    , São Paulo      , 23    , 59.0
12, Janaina  , Rio de Janeiro , 32    , 86.0
13, Wong     , Brasília       , 43    , 89.0
14, Roberto  , Salvador       , 38    , 98.0
15, Marco    , Curitiba       , 31    , 61.0
16, Paula    , Belo Horizonte , 34    , 44.0
```


usamos o arquivo dos alunos para ler as idades e as notas, que são convertidas para inteiro e flutuante, respectivamente.

```
» with open('alunos.txt', 'r') as f:
»     maior, nota_media, n = 0, 0, 0
»     f.readline()
»     for linha in f:
```

```
»         palavras = linha.split(',')
»         n +=1
»         nota_media += float(palavras[4])
»         idade = int(palavras[3])
»         if idade > maior:
»             maior = idade
»             nome = palavras[1]
» print('%s é o aluno mais velho, com %d anos.' % (nome.strip(), maior))
» print('A nota média entre %d alunos é %2.1f' % (n, nota_media/n))

↳ Wong é o aluno mais velho, com 43 anos.
↳ A nota média entre 7 alunos é 74.3
```

O método `split(',')` quebra a linhas em palavras, partindo cada uma nas vígulas e retornando uma lista. `palavra[3]` é a idade, `palavra[4]` a nota. A maior idade e o nome do aluno mais velho são armazenados dentro do teste final.

# Propriedades de arquivos

`writelines()` pode receber uma lista contendo linhas de texto. No exemplo abaixo um novo arquivo com 3 linhas é gravado.

```
» lista = ['Esta é a linha 1\n',
»          'Esta é a linha 2\n',
»          'Esta é a linha 3'
»         ]
» with open('novo.txt', 'w') as f:
»     f.writelines(lista)
» with open('novo.txt', 'r') as f:
»     print(f.read())

↳ Esta é a linha 1
↳ Esta é a linha 2
↳ Esta é a linha 3
```

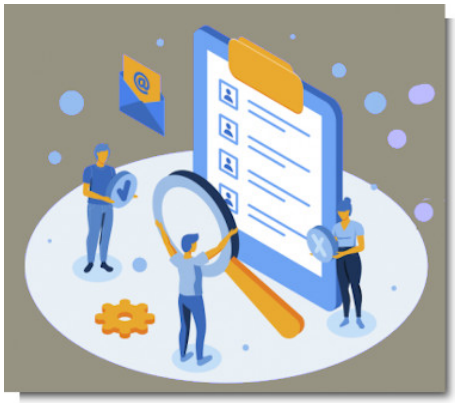
Um objeto de arquivo tem diversas propriedades, entre elas `file.name` , o nome do arquivo, `file.closed` que é `True` se o arquivo está fechado (embora a variável continue apontando para o objeto), e `file.mode` que contém o modo em que ele foi aberto.

```
» with open('novo.txt', 'r') as f:
»     info = 'O arquivo "'+ f.name + '" está ' + ('fechado' if f.closed else 'aberto')
»     info += ' no modo %s' % f.mode
»     print(info)
» print('Agora o arquivo está ' + ('fechado' if f.closed else 'aberto'))

↳ O arquivo "novo.txt" está aberto no modo r
↳ Agora o arquivo está fechado
```

Tradutor

# Leitura incremental



A abertura de um arquivo com muitas linhas, ou uma linha muito longa sem quebras pode esgotar os recursos de memória de seu computador. Para evitar esse problema tanto `readline` ou `readlines` aceitam um argumento opcional estabelecendo quanto dado deve ser lido de cada vez. É possível ler um arquivo de modo incremental com `arquivo.read(n)` onde `n` é o número de bytes lidos (que resulta em um caracter simples). Como a primeira linha contém 16 caracteres ('Esta é a linha 1') ela é esgotada em duas iterações.

```
» with open('novo.txt', 'r') as f:
»     print(f.read(8))
»     print(f.read(8))

↳ Esta é a
↳  linha 1
```

Após as 2 leituras no código anterior o ponteiro (cursor) está posicionado sobre o byte 17. Essa posição pode ser lida e alterada, usando-se os métodos `tell()` e `seek()`. Nas linhas abaixo 17 bytes são lidos e impressos na linha 1. A linha 2 mostra a posição do ponteiro, `18`, obtida com `seek()`.

```
» with open('linhas.txt', 'r') as f:
»     print('1 : ', f.read(17), end='')
```

```
»     print('2 : ', f.tell())
»     f.seek(0, 0)
»     print('3 : ', f.read(17), end='')
»     f.seek(36, 0)
»     print('4 : ',  f.read(17), end='')

↳ 1 :  Esta é a linha 1
↳ 2 :  18
↳ 3 :  Esta é a linha 1
↳ 4 :  Esta é a linha 3
```



No exemplo seguinte a primeira linha é lida inteira e apenas 10 caracteres da segunda linha.

```
» with open('alunos.txt', 'r') as f:
»     txt1 = f.readline()
»     txt2 = f.readline(10)
» print(txt1, end = '')
» print(txt2)

↳ id, nome     , cidade         , idade , nota
↳ 10, Pedro    , São Paulo
```

# Método `open()`

| Método | Descrição |
|---|---|
| open(arquivo, mode) | Abre o arquivo (descrição dos parâmetros abaixo) |

Os seguintes parâmetros são usados com `open()` :

| Parâmetro | Descrição |
|---|---|
| arquivo | caminho completo e nome do arquivo |
| mode | Uma string que define em que modo o arquivo será aberto |

O parâmetro `mode` pode ser:

| Parâmetro | Descrição |
|---|---|
| `r` | para leitura – O ponteiro é colocado no início do arquivo. Default. |
| `r+` | para leitura e gravação. O ponteiro fica no início do arquivo. |
| `w` | apenas para gravação. Substitui arquivo existente, cria novo se o arquivo não existir. |

| | |
|---|---|
| `w+` | escrita e leitura. Substitui arquivo existente, cria novo se arquivo não existir. |
| `rb` | para leitura em formato binário. O ponteiro fica no início do arquivo. |
| `rb+` | para leitura e escrita em formato binário. |
| `wb+` | para escrita e leitura em formato binário. Substitui o arquivo existente. Cria novo se não existir. |
| `a` | para anexar. Conteúdo existente fica inalterado e o ponteiro fica no final do arquivo. Cria novo se não existir. |
| `ab` | um arquivo para anexar em formato binário. O ponteiro fica no final do arquivo. Cria novo se não existir. |
| `a+` | para anexar e ler. O ponteiro fica no final do arquivo, se arquivo existir. Cria novo se não existir. |
| `ab+` | anexar e ler em formato binário. O ponteiro no final do arquivo. Cria novo se não existir. |
| `x` | cria novo arquivo lançando erro se já existir. |

# São métodos do objeto `file`

| Método | Descrição |
|---|---|
| `close()` | fecha o arquivo; sem efeito se arq. já está fechado |
| `detach()` | retorna o fluxo bruto separado do buffer |
| `fileno()` | retorna número inteiro diferente para cada arquivo aberto |
| `flush()` | descarrega no disco as alterações no *buffer* |
| `isatty()` | retorna `True` se o fluxo de arquivo é interativo |
| `next(arquivo)` | itera sobre arquivo, lançando erro no final |
| `read()` | retorna o conteúdo do arquivo |
| `readable()` | retorna `True` se o fluxo do arquivo pode ser lido |
| `readline()` | retorna uma única linha do arquivo |
| `readlines()` | retorna lista com todas as linhas do arquivo |
| `search()` | localiza item no arquivo e retorna sua posição |
| `searchable()` | retorna `True` se o arquivo é pesquisável, por ex. com seek() |
| `tell()` | retorna a posição atual do arquivo |
| `truncate([tamanho])` | redimensiona (truncando) o arquivo para um tamanho especificado |
| `writable()` | retorna `True` se o arquivo pode receber gravações |
| `write()` | grava a string especificada no arquivo |
| `writelines()` | escreve uma sequência no arquivo. Qualquer objeto iterável composto por strings pode ser usado |

# Módulo os


Vimos que existem diversas funções e outros objetos no módulo básico que são carregados juntamente com o próprio Python. No entanto muitos outros módulos podem sem anexados no código, inclusive aqueles da chamada biblioteca padrão. Também existem módulos desenvolvido por terceiros que podem ser baixados e usados. Além disso o usuário pode criar seus próprios módulos e importá-los em seus aplicativos.



De particular interesse para a manipulação de arquivos é o módulo `os` que contém as funcões básicas de interação com o sistema operacional. Para habilitar o uso de módulos instalados usamos `import nome_do_modulo` .

# Listar pastas e arquivos

Por exemplo, após a importação de `os` podemos usar o método `os.listdir(pasta)` que retorna uma lista com o conteúdo da pasta especificada.

```
» import os
» pasta ='/home/guilherme/Projetos/Python'
» lista_arquivos = os.listdir(pasta)
» print(lista_arquivos)
↳ ['arquivos', 'wallpaper.py', 'turtle.py', 'numpy_02.py', 'NLP', 'output', 'osDir.py',
↳  'lerShelfFile.py', 'guessNumber.py', ...]
```

Embora esse método ainda esteja disponível, no Python 3.5 foi inserido uma nova forma de obter esse resultado por meio de `os.scandir()` que retorna um iterável. Os elementos desse iterável possuem diversas propriedades, entre elas `name` , usada para recuperar o nome do arquivo ou pasta.

```
» arquivos = os.scandir(pasta)
» for arquivo in arquivos:
»     print(arquivo.name)
↳ wallpaper.py
↳ turtle.py
↳ numpy_02.py
```

```
↳ NLP
↳ output
↳ osDir.py
↳ ...
```

**Observação (Windows)**: Como o caracter `\` tem significado especial (escape) no Python é necessário escrever caminhos no Windows de uma forma especial, de uma das duas formas:



```
» caminho = r'C:\Windows\Temp'
# ou
» caminho = 'C:\\Windows\\Temp'
```

No código abaixo usamos `os.path.join(local, t)` que faz a concatenação correta do caminho com o nome do arquivo, agindo de forma diferente de acordo com o sistema operacional. O método `os.path.split` age de forma inversa, retornando uma tupla que contém a caminho e o nome do arquivo. Depois de extraído o nome do arquivo podemos obter uma tupla com nome separado da extensão usando `os.path.splitext` .

```
# como funciona os.path.join (no linux ou OSX)
» os.path.join('caminho_completo', 'nome_do_arquivo')
↳ 'caminho_completo/nome_do_arquivo'

# separando caminho do arquivo
» os.path.split('/home/usuario/Documentos/Artigo.txt')
↳ ('/home/usuario/Documentos', 'Artigo.txt')

# separando a extensão
» os.path.splitext('image.jpeg')
↳ ('image', '.jpeg')

# no windows uma string diferente seria retornada com join
» os.path.join('caminho_completo', 'nome_do_arquivo')
↳ 'caminho_completo\\nome_do_arquivo'
```

# Filtrar pastas e arquivos

Para investigar se o elemento retornado por `listdir` é um arquivo podemos usar `os.path.isfile()` , e `os.path.isdir()` para pastas (diretórios).

```
# para filtrar os arquivos
» local = '.'
» for t in os.listdir(local):
»     if os.path.isfile(os.path.join(local, t)):
»         print(t)
↳ arquivo001.txt
↳ arquivo002.txt
↳ jupyter001.ipynb
↳ teste.csv

# para filtrar as pastas
» for t in os.listdir(local):
»     if os.path.isdir(os.path.join(local, t)):
»         print(t)
↳ .ipynb_checkpoints
↳ dados

# para listar arquivos dentro da pasta 'dados'
» local = './dados'
» for t in os.listdir(local):
»     if os.path.isfile(os.path.join(local, t)):
»         print(t)
↳ nums.csv
↳ alunos.pkl
```

A mesma operação feita usando `os.scandir()`

```
# usando scandir
» local = '.'
» with os.scandir(local) as arqs:
»     for arq in arqs:
»         if arq.is_file():
»             print('arquivo:', arq.name)
»         elif arq.is_dir():
»             print('pasta:  ', arq.name)
↳ arquivo: pandas001.ipynb
↳ arquivo: python001.ipynb
↳ arquivo: teste_gravar
↳ arquivo: alunos.txt
↳ arquivo: teste_novo.csv
↳ pasta:   .ipynb_checkpoints
↳ arquivo: teste.py
↳ arquivo: alunos.csv
↳ pasta:   dados
↳ ...
```

Claro que essa mesma lista pode ser obtida através de uma compreensão de lista, o que ilustra mais uma vez o poder de concisão dessa construção.

```
# para arquivos
» lst_arquivos = [t.name for t in os.scandir(local) if t.is_file()]
» lst_arquivos
↳ [pandas001.ipynb, python001.ipynb, teste_gravar, alunos.txt,
↳  teste_novo.csv, teste.py, alunos.csv]

# para as pastas
» [t.name for t in os.scandir(local) if t.is_dir()]
↳ ['.ipynb_checkpoints', 'dados']
```

A recuperação dos nomes dos arquivos, associada a testes de string com esses nomes, permite uma filtragem mais específica de arquivos retornados. No exemplo abaixo usamos `.endswith` para verificar se os arquivos possuem uma extensão determinada (no caso '.csv'). No segundo exemplo usamos compreensão de lista e testamos a substring após o último ponto.

```
# usando .endswith()
» print('Arquivos com extensão csv:')
» with os.scandir(local) as arqs:
»     for arq in arqs:
»         if arq.is_file() and  arq.name.endswith('.csv'):
»             print(arq.name)
Arquivos com extensão csv:
↳ teste_novo.csv
↳ alunos.csv

# alternativamente, usando compreensão de lista e .split('.')
» [arq.name for arq in os.scandir(local) if arq.is_file() if arq.name.split('.')[-1]=='txt']
↳ ['alunos.txt']
```

O método `os.scandir()` não recupera apenas os nomes dos arquivos. Cada objeto do *ScandirIterator* possue o método `.stat()` que retorna informações sobre o arquivo ou pasta ao qual se refere, como tamanho do arquivo. O atributo `st_mtime`, por ex., contém a hora da última alteração.

No exemplo seguinte fazemos uma iteração sobre os arquivos e pastas em `local` e imprimimos a data de sua última alteração.

```
# atributo de arquivos: tempo desde última alteração
» with os.scandir(local) as arquivos:
»     for arq in arquivos:
»         info = arq.stat()
»         print(info.st_mtime)
↳ 1612654770.8688447
↳ 1615833776.4532673
↳ 1611327215.6848917
↳ ...
```



O retorno é dado em *segundos desde a **época***, uma medida de tempo também chamada de *Unix time* que consiste no número de segundos decorridos desde Zero Horas do dia 1 de Janeiro de 1970 (UTC, longitude 0°) e pode ser convertido em uma data mais legível através de diversas funções de tratamento de datas e horas.

No código seguinte lemos os arquivos na pasta `local` e retornamos seus nomes e datas de última modificação, convertidos em forma mais legível usando uma função do módulo `datetime`. Veremos mais tarde outras funcionalidades desse módulo.

```
» from datetime import datetime

» def converter_data(timestamp):
»     d = datetime.utcfromtimestamp(timestamp)
»     return d.strftime('%d/%m/%Y')

» def ler_arquivos():
»     arquivos = os.scandir(local)
»     for arq in arquivos:
»         if arq.is_file():
»             info = arq.stat()
»             print(f'{arq.name}\t\t Modificado em: {converter_data(info.st_mtime)}')

↳ Pandas001.ipynb      Modificado em: 06/02/2021
↳ Python001.ipynb      Modificado em: 15/03/2021
↳ teste_gravar         Modificado em: 15/03/2021
↳ alunos.txt           Modificado em: 16/03/2021
↳ teste_novo.csv       Modificado em: 16/11/2020
↳ ...
```

Os argumentos passados para `strftime()` são: `%d` o dia do mês, `%m` o número do mês e `%Y` o ano, com 4 dígitos. Várias outras formatações são possíveis.

O módulo `os` contém os métodos `os.getcwd()` para ler a pasta ativa no momento, e `os.chdir('nova_pasta')` para trocar para uma nova pasta.

```
» import os
» os.getcwd()
↳ '/home/guilherme/Projetos/Phylos.net'

» os.chdir('/home/guilherme/Music')
» os.getcwd()
↳ '/home/guilherme/Music'
```

Também podemos criar novas pastas. Para isso usamos `os.mkdir()` (cria uma pasta) e `os.makedirs()` (cria várias pastas). Podemos criar, dentro da pasta atual, a pasta `exemplo` e `exemplo/textos` .

```
» import os
» os.mkdir('exemplo')
» os.mkdir('exemplo/textos')

# a tentativa de criar uma pasta existente resulta em erro
» os.mkdir('exemplo')
↳ FileExistsError: [Errno 17] File exists: 'exemplo'
```

Várias pastas e subpastas podem ser criadas simultaneamente. O comando abaixo cria três pastas.

```
» os.makedirs('2021/03/22')
```

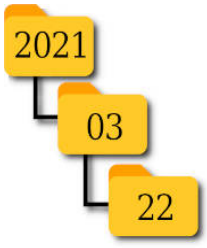


Figura 1

A estrutura de pastas criada na pasta atual é ilustrada na figura 1. Os seguintes comandos ilustram operações feitas com métodos de `os` :

```
# armazena pasta de trabalho atual, antes da modificação
» pasta_trabalho = os.getcwd()
# altera pasta atual para outra
» os.chdir('/home/guilherme/Music')
# cria subpastas na pasta atual (resultado semelhante ao da figura 1)
» os.makedirs('subpasta10/subpasta11/subpasta12')

# retorna para a pasta de trabalho
» os.chdir(pasta_trabalho)
```

```
# cria sub pasta
» os.mkdir('teste')

# muda pasta atual ('.' é atalho para pasta atual)
» os.chdir('./teste')
# verifica qual é a pasta atual
» os.getcwd()
↳ '/home/guilherme/Projetos/Artigos/teste'

# volta para pasta no nível acima ('..' é atalho para pasta 'mãe')
» os.chdir('..')

# renomeia pasta
» os.rename('teste', 'novo')

# os.remove só apaga arquivos
» os.remove('novo') # só pode apagar arquivo
↳ IsADirectoryError: [Errno 21] Is a directory: 'novo'

» os.remove('aluno.txt') # só pode apagar arquivo
# o arquivo 'aluno.txt' foi apagado (se existe)

# para apagar uma pasta use rmdir
» os.rmdir('novo')
# a pasta 'novo' foi apagada
```

O método `os.remove(arquivo)` lança um erro se o arquivo não existe ou se é uma pasta. Para evitar essa possibilidade usamos um `try` ou testamos previamente o arquivo. Para apagar pastas e subpastas podemos percorrer cada uma delas (veja método abaixo) ou usar o módulo `shutil`.

```
» arquivo_apagar = 'home/data.txt'
# testa a existência da arquivo
» if os.path.isfile(arquivo_apagar):
»     os.remove(arquivo_apagar)
» else:
»     print(f'O arquivo: {data_file} não existe ou é uma pasta')
```

Para apagar uma pasta podemos usar `os.rmdir()` ou `shutil.rmtree()`.

```
» apagar_pasta = 'documentos/pasta'
» try:
»     os.rmdir(apagar_pasta)
» except OSError as e:
»     print(f'Error: {apagar_pasta} : {e.strerror}')
```

```
# apagar pastas e subpastas, mesmo que não estejam vazias
» import shutil
» apagar_pasta = 'documentos/pasta'
» try:
»     shutil.rmtree(apagar_pasta)
» except OSError as e:
»     print(f'Error: {apagar_pasta} : {e.strerror}')
```

Tradutor

# Percorrendo a árvore de pastas

Não é rara a necessidade de percorrer as pastas e subpastas em uma estrutura de diretórios, eventualmente executando uma tarefa nos arquivos no disco ou fazendo buscas. O método `os.walk()` pode auxiliar nessa tarefa, percorrendo a árvore tanto da pasta raiz para as subpastas (de cima para baixo, *top down*) quanto no sentido inverso (*bottom up*). Por *default* `os.walk(pasta)` faz várias iterações em pasta e subpastas, até esgotar a árvore, de mãe para filhos. Em cada iteração retorna:

- uma string com o nome da pasta atual
- uma lista de suas subpastas
- uma lista dos arquivos da pasta atual

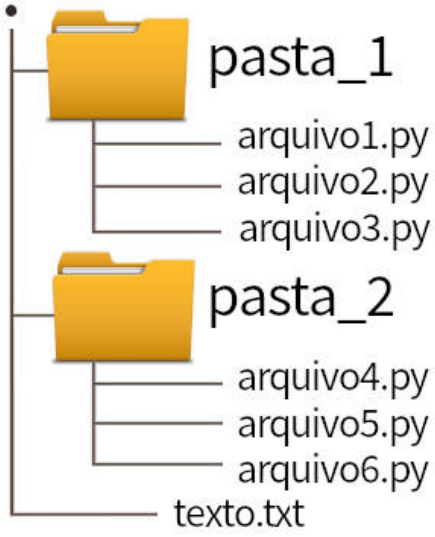



Figura 2

Depois a iteração passa para uma subpasta (no modo de cima para baixo) ou para a pasta mãe. Para percorrer a árvore no sentido de baixo para cima usamos o parâmetro `os.walk(pasta, topdown=False)`. No código que se segue percorremos a árvore (lembrando que `'.'` simboliza pasta ativa atual). O teste `if not sub_pastas` resulta `True` se a lista está vazia. O resultado exibido supõe uma estrutura de pastas como na figura 2.

```
# usando os.walk() para listar arquivos e pastas
» for pasta, sub_pastas, arquivos in os.walk('.'):
»     if not sub_pastas:
»         print('A pasta:', pasta, 'não possui
subpastas')
»     else:
»         print('A pasta:', pasta, 'possui as
subpastas:')
»         for sub in sub_pastas:
»             print('\t\t',sub)
»     if not arquivos:
```

```
»         print('\t não possui arquivos')
»     else:
»         print('\t e os arquivos:')
»         for nome_arquivo in arquivos:
»             print('\t\t', nome_arquivo)

↳ A pasta: . possui as subpastas:
↳       pasta_1
↳       pasta_2
↳    e os arquivos:
↳       texto.txt
↳ A pasta: ./pasta_1 não possui subpastas.
↳    e os arquivos:
↳       arquivo1.py
↳       arquivo2.py
↳       arquivo3.py
↳ A pasta: ./pasta_2 não possui subpastas.
↳    e os arquivos:
↳       arquivo4.py
↳       arquivo5.py
↳       arquivo6.py
```



# Informações do sistema operacional

O módulo `os` possui muitos outros métodos. Por exemplo, `os.uname()` retorna o sistema operacional, nome de usuário e dados da versão do sistema usado.

```
» for t in os.uname():
»     print(t)
↳ Linux
↳ guilherme-Lenovo
↳ 5.8.0-45-generic
↳ #51-Ubuntu SMP Fri Feb 19 13:24:51 UTC 2021
↳ x86_64
```

O módulo **os** possui algumas propriedades úteis, entre elas `os.environ` , que contém informação sobre o sistema operacional e seu ambiente (*environment*), armazenadas em um dicionário.

```
» os.system
↳ <function posix.system(command)>
```

```
» os.environ
↳ environ{'QT_SCALE_FACTOR': '1',
        'LANGUAGE': 'en_US',
        'SHELL': '/bin/bash',
        'LC_NUMERIC': 'pt_BR.UTF-8',
        'LC_PAPER': 'pt_BR.UTF-8',
        'MATE_DESKTOP_SESSION_ID': 'this-is-deprecated',
        'JPY_PARENT_PID': '9099',
        'TERM': 'xterm-color',
        ... [truncado]
        'MPLBACKEND': 'module://ipykernel.pylab.backend_inline'}

» # essa propriedade é um tipo dicionário. por ex.:
» os.environ['USER']
↳ guilherme
```

Os seguintes testes foram realizados no *prompt* do python, em sessão do Anaconda.

```
>>> import os
>>> os.times()
↳ posix.times_result(user=0.03, system=0.0, children_user=0.0, children_system=0.0,
elapsed=17180795.37)
>>> os.system('date')
↳ sáb 05 jun 2021 14:19:27 -03

# supondo a existência de um programa (ou atalho) de nome caja
# o aplicativo caja é executado (no caso um gerenciador de arquivos)
>>> os.system('caja')
```

O método `os.walk(top, topdown=True, onerror=None, followlinks=False)` retorna tuplas com pasta atual, subpastas e arquivos nelas contidos. Por *default* ele percorre as pastas iniciando em `top` e descendo para as substastas nele localizadas.

```
» import os
» path = '/home/guilherme/Temp'
» for (root, dirs, files) in os.walk(path):
»     print('Pasta atual: %s' % root)
»     print('Com as subpastas:\n', dirs)
»     print('Arquivos presentes:\n', files)
»     print('\n--------------------------------\n')

# saida (truncada)
↳ Pasta atual: /home/guilherme/Temp
↳ Com as subpastas:
↳  ['RevelationSpace', 'pandas']
```

```
↳ Arquivos presentes:
↳  ['Livro1', 'modal.html', 'FileZilla.xml']
↳
↳ Pasta atual: /home/guilherme/Temp/RevelationSpace
↳ Com as subpastas:
↳  []
↳ Arquivos presentes:
↳  ['01-RevealSpace.mp3', '02-RevealSpace.mp3', '03-RevealSpace.mp3', '04-RevealSpace.mp3', '05-
RevealSpace.mp3']
↳
↳ Pasta atual: /home/guilherme/Temp/pandas
↳ Com as subpastas:
↳  ['Lambda', 'Pandas']
↳ Arquivos presentes:
↳  ['Lambda.html', 'Pandas.html']
```



# Métodos de OS

Segue uma lista de alguns dos métodos do módulo `os` :

| Método | significado |
|---|---|
| `chdir("novaPasta")` | muda a pasta ativa para novaPasta. Se novaPasta = ".." vai para pasta mãe |
| `os.environ()` | obtenha o ambiente dos usuários, |
| `os.getcwd()` | retorna pasta de trabalho atual, |
| `os.getgid()` | retorna id de grupo real do processo atual, |
| `os.getuid()` | retorna ID do usuário do processo atual, |
| `os.getpid()` | retorna ID de processo real do processo atual, |
| `os.umask(mascara)` | define o umask numérico atual e retorne o umask anterior, |
| `os.uname()` | retorna informações que identificam o sistema operacional atual, |
| `os.chroot(caminho)` | altera pasta raiz do processo atual para caminho, |
| `os.listdir(caminho)` | retorna lista de arquivos e pastas no caminho, |
| `os.mkdir(caminho)` | cria pasta *caminho*, |
| `os.makedirs(sequencia)` | criação recursiva de pastas na sequencia, |
| `os.remove(caminho_arquivo)` | remove arquivo em *caminho_arquivo*, |
| `os.removedirs(caminho)` | remove pastas recursivamente, |
| `os.rename('nomeAntigo','nomeNovo')` | renomeia arquivo/pasta *'nomeAntigo'* para *'nomeNovo'dst*, |
| `os.rmdir(caminho)` | remove pasta em *caminho*, |
| `os.system(comando)` | executa *comando* na shell do sistema operacional, |

| | |
|---|---|
| `os.uname()` | retorna dados sobre o sistema operacional, usuário e máquina usada |
| `os.walk(caminho)` | retorna tuplas com pasta, subpastas e arquivos em cada uma. |

Continue a leitura Classes, variáveis do Usuário

Tradutor

# Bibliografia


- Python Documentation: os — Miscellaneous operating system interface.


Consulte a bibliografia no final do primeiro artigo dessa série.

Posted in Programação, python

Tagged arquivos, os, pastas, Python


← Python: Compreensão de listas

Python: Classes, variáveis do usuário →


## Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Tradutor

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment

## Categorias de Temas

- Ceticismo
- Ciência
- Contos
- Cosmologia
- Debate
- Django
- Ficção Científica
- Física
- História da Matemática
- Inteligência Artificial
- Matemática
- Matemática Básica
- Matemática Superior
- Notícias
- NumPy
- Pandas

Programação
python
Science Fiction
Sem Categoria
SQL
SQLAlchemy
Tecnologias WEB
Uncategorized
Web



## Recent Posts

ORM e Relacionamentos
SQLAlchemy: ORM
SQLAlchemy: UPDATE e DELETE
SQLAlchemy: INSERT e SELECT
SQLAlchemy: Agrupamentos e Subqueries

Search ... Search