**Como fazer uma cópia de uma lista no Python**

* 6 de setembro de 2018
* /
* [Programação](https://blog.alura.com.br/category/programacao/) [Python](https://blog.alura.com.br/category/programacao/python/)
* /
* [0 Comments](https://blog.alura.com.br/como-fazer-copia-de-lista-python/#disqus_thread)

***(Last Updated On: 22 de agosto de 2018)***

Eu e meus amigos do clube de livros criamos um sistema em Python para organizar que livros cada um de nós tem.

Cada um tem sua própria lista, na qual os livros são divididos por categoria (no meu caso, **SQL**, [**PHP**](https://blog.alura.com.br/glossario/php/) e **Front-end**). Cada categoria também é uma lista:

livros\_yan = [[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

Copy

Recentemente, recebemos um participante novo do clube, Pedro, e ele decidiu comprar uma cópia de todos os livros que eu tinha:

livros\_pedro = livros\_yan

print(livros\_pedro)

Copy

E agora a lista de livros do Pedro:

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

Tudo certo!

Seguimos o clube e eu acabei adquirindo outro livro de uma categoria que eu ainda não tinha – **Games**:

livros\_games = [‘Jogos iOS’]

livros\_yan.append(livros\_games)

print(livros\_yan)

Copy

Agora olha como está minha lista:

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’],

[‘Jogos iOS’]]

Certo, como esperávamos! Acontece que o Pedro foi checar a lista de livros dele:

print(livros\_pedro)

Copy

Mas olha qual foi o resultado:

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’],

[‘Jogos iOS’]]

Ué! Mas ele não tinha comprado o livro **Jogos iOS**! A lista dele ficou como a lista dos meus livros… Por que será?

**Listas são objetos mutáveis**

Por que quando alteramos a minha lista, a lista do Pedro foi alterada junto? O que acontece é que não eram listas separadas, **eram a mesma lista**.

Quando atribuímos como valor de uma variável a referência de uma outra, o que estamos fazendo é apontar as duas variáveis, ou seja, os dois nomes, para o mesmo objeto na memória do computador. Podemos comprovar isso com nosso conhecido [operador de identidade is](https://blog.alura.com.br/qual-a-diferenca-entre-e-is-no-python):

livros\_pedro = livros\_yan

print(livros\_pedro is livros\_yan)

Copy

E o resultado:

True

O mesmo acontece com qualquer outro tipo no Python, como strings:

meu\_nome = ‘Yan’

copia\_nome = meu\_nome

print(meu\_nome is copia\_nome)

Copy

O resultado:

True

E o que acontece se tentarmos alterar uma dessas variáveis?

meu\_nome += ‘ Orestes’

print(meu\_nome)

print(copia\_nome)

Copy

Olha o resultado:

Yan Orestes

Yan

Só alterou a variável que pedimos para alterar! Mas ué, não era o mesmo objeto? **Era, mas deixou de ser quando fizemos essa alteração**. Olha só:

meu\_nome = ‘Yan’

copia\_nome = meu\_nome

print(‘Antes da alteração:’)

print(id(meu\_nome))

print(id(copia\_nome))

meu\_nome += ‘ Orestes’

print(‘Depois da alteração:’)

print(id(meu\_nome))

print(id(copia\_nome))

Copy

E o resultado:

Antes da alteração:

139683901853232

139683901853232

Depois da alteração:

139683901897776

139683901853232

Note que o identificador da variável meu\_nome mudou assim que alteramos seu valor. Isso é porque **strings, assim como grande parte dos tipos nativos do Python, são imutáveis**.

“*Mas como imutáveis, se acabamos de mudar o valor de uma string?*”

Apesar de parecer que alteramos a string, na verdade o que alteramos foi apenas a referência que era guardada no nome de variável meu\_nome.

Não alteramos o objeto string original, porque **não podemos fazer isso**, mas criamos outro objeto string com valor Yan Orestes (ou meu\_nome + ‘ Orestes’) e mudamos a referência em nossa variável. Por isso, o ID mudou.

Diferente das string, **listas são mutáveis**. Ou seja, podemos alterar um objeto lista mantendo ele no mesmo espaço na memória, em vez de termos de criar outro objeto para substituir a referência da variável.

Assim, quando usamos o método clear(), ainda estamos tratando do mesmo objeto lista com os dois identificadores:

livros\_yan = [[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

livros\_pedro = livros\_yan

print(‘Antes da alteração:’)

print(id(livros\_yan)

print(id(livros\_pedro))

livros\_games = [‘Jogos iOS’]

livros\_yan.append(livros\_games)

print(‘Depois da alteração:’)

print(id(livros\_yan))

print(id(livros\_pedro))

Copy

E o resultado:

Antes da alteração:

139715027862984

139715027862984

Depois da alteração:

139715027862984

139715027862984

O identificador das duas variáveis se manteve o mesmo, pois continuamos tratamos do mesmo objeto mesmo depois de sua alteração. Como, então, podemos criar um segundo objeto lista igual a um já existente?

**Fazendo a cópia de uma lista**

Os tipos sequenciais no Python, como as lista, nos disponibilizam uma técnica que pode nos ajudar em nosso objetivo – o fatiamento, ou [*slicing*](https://docs.python.org/3/tutorial/introduction.html#lists). O *slicing* nos permite criar um **outro** objeto apenas com um pedaço desejado do objeto original.

A sintaxe do *slicing* é parecida com a de uma indexação comum, mas com um : separando o primeiro elemento que queremos do último, **sendo o último descartado**. Com listas, podemos fazer algo como o seguinte:

>>> livros\_yan = [[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’],

... [‘HTML5 e CSS3’]]

>>> livros\_yan[0:2]

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’]]

Copy

Assim, podemos ter uma cópia da lista em outro objeto:

livros\_yan = [[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

livros\_pedro = livros\_yan[0:3]

print(id(livros\_yan))

print(id(livros\_pedro))

Copy

E os IDs:

139715027878216

139715052415952

O Python ainda nos permite omitir o primeiro número do *slicing* se quisermos pegar desde o começo, e o segundo se quisermos até o final, o que simplifica nosso código:

livros\_pedro = livros\_yan[:]

print(livros\_pedro)

Copy

E o resultado:

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

Os mesmos da minha lista!

Agora vamos tentar adicionar uma categoria à minha lista de livros e ver se a do Pedro muda também:

livros\_games = [‘Jogos iOS’]

livros\_yan.append(livros\_games)

print(livros\_yan)

print()

print(livros\_pedro)

Copy

Dessa vez:

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’],

[‘Jogos iOS’]]

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

Deu certo! Também podemos usar o método de lista copy(), que tem o mesmo comportamento:

livros\_pedro = livros\_yan.copy()

Copy

E continua funcionando da mesma maneira!

**O problema das cópias rasas**

Todo o nosso sistema estava funcionando bem, até que eu consegui tirar a certificação PHP e decidi doar meu livro sobre o assunto para um amigo, criando a necessidade de remover esse livro de minha lista:

livros\_yan[1].remove(‘Certificação PHP’)

print(livros\_yan)

Copy

E agora, minha lista:

[[‘Banco MySQL’], [‘TDD PHP’], [‘HTML5 e CSS3’], [‘Jogos iOS’]]

De novo, o Pedro foi checar a lista dele:

print(livros\_pedro)

Copy

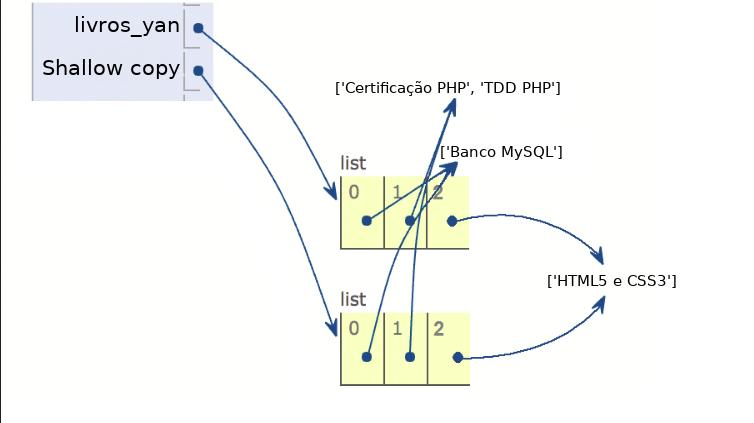
E o resultado:

[[‘Banco MySQL’], [‘TDD PHP’], [‘HTML5 e CSS3’]]

O livro sumiu para ele também! Repare que nossas listas de livros ainda estão diferentes, mas quando eu removi um livro da categoria **PHP**, na lista dele também foi removido. O que acontece?

Quando usamos o método .copy() ou o *slicing* para copiar uma lista, estamos fazendo uma **cópia rasa**, ou, tecnicamente, uma [*shallow copy*](https://en.wikipedia.org/wiki/Object_copying#Shallow_copy).

Em uma lista, a cópia rasa vai criar outro objeto lista para armazenar os mesmos valores da primeira lista, **mas vai usar os mesmos objetos que a primeira lista em seu interior**, como indica a imagem:

[](https://blog.alura.com.br/wp-content/uploads/2018/04/diagrama-shallow.png)

Podemos até checar os IDs dos membros das listas:

livros\_yan = [[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

livros\_pedro = livros\_yan.copy()

print(‘livros\_yan - ID: {}’.format(id(livros\_yan))

print(‘livros\_pedro - ID: {}’.format(id(livros\_pedro))

print()

print(‘livros\_yan[1] - ID: {}’.format(id(livros\_yan[1]))

print(‘livros\_pedro[1] - ID: {}’.format(id(livros\_pedro[1]))

sao\_mesmo\_objeto = livros\_yan[1] is livros\_pedro[1]

print(‘livros\_yan[1] is livros\_pedro[1]: {}’.format(sao\_mesmo\_objeto))

Copy

E o resultado:

livros\_yan - ID: 139747348710920

livros\_pedro - ID: 139848030606832

livros\_yan[1] - ID: 139747372553152

livros\_pedro[1] - ID: 139747372553152

livros\_yan[1] is livros\_pedro[1]: True

Como podemos ver, apesar do método copy() ter criado um outro objeto lista para armazenar os valores, todos os elementos da lista livros\_yan foram reutilizados.

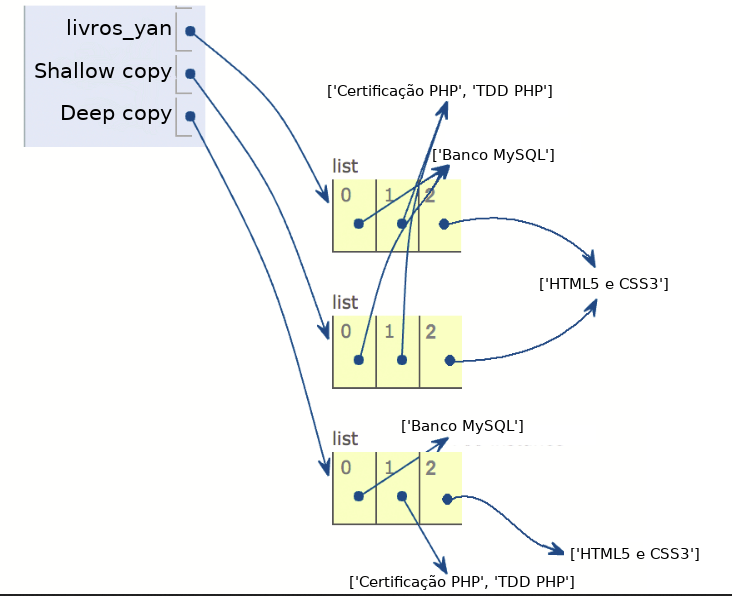
Dessa forma, temos o mesmo problema do começo, já que listas são objetos mutáveis. Isso é o que limita as cópias rasas. Como podemos superar isso?

**Fazendo uma cópia profunda de uma lista**

Em programação, ainda temos outro conceito de cópia, que se diferencia da cópia rasa – a **cópia profunda**, ou [*deep copy*](https://en.wikipedia.org/wiki/Object_copying#Deep_copy).

Diferentemente da cópia rasa, ela não simplesmente insere as referências dos valores da lista original dentro a nova lista, mas também cria novos objetos para cada elemento, de fato separando uma lista da outra.

A imagem abaixo complementa a imagem anterior com esse novo conceito:

[](https://blog.alura.com.br/wp-content/uploads/2018/04/diagrama-shallow-deep.png)

Mas como podemos usar esta técnica no Python? Será que teremos que implementar nós mesmos uma função que faça isso recursivamente? Parece trabalhoso…

Por conta desse problema recorrente de objetos compostos, como nosso caso de listas que incluem outras listas, o Python já vem nativamente com uma solução em sua biblioteca padrão – com o módulo [**copy**](https://docs.python.org/3.6/library/copy.html).

Dentro desse módulo, temos uma função específica para isso – a deepcopy(). Importando-a, seu uso é objetivo:

from copy import deepcopy

livros\_yan = [[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

livros\_pedro = deepcopy(livros\_yan)

Copy

Agora vamos tentar modificar a lista livros\_yan:

livros\_games = [‘Jogos iOS’]

livros\_yan.append(livros\_games)

livros\_yan[1].remove(‘Certificação PHP’)

print(livros\_yan)

print(livros\_pedro)

Copy

Dessa vez:

[[‘Banco MySQL’], [‘TDD PHP’], [‘HTML5 e CSS3’], [‘Jogos iOS’]]

[[‘Banco MySQL’], [‘Certificação PHP’, ‘TDD PHP’], [‘HTML5 e CSS3’]]

Certo! Agora sim funcionou exatamente como queríamos!

**Conclusão**

Nesse post, pudemos entender a diferença de objetos mutáveis e imutáveis no Python, com exemplos usando listas. Vimos, então, uma forma de não nos atrapalharmos com isso com o método copy() ou o *slicing* de lista.

Assim, aprendemos sobre **cópia rasa** e seus possíveis problemas para com objetos compostos, e como resolver isso com **cópias profundas**.

Agora sabemos quando basta uma cópia rasa, como em listas comuns, e quando precisamos de cópias profundas, como em listas que contêm outras listas.

E aí, gostou do conteúdo? Se quiser conhecer outras funcionalidades e peculiaridades de listas no Python, dê uma olhada em nossos posts sobre [adicionar novos elementos em uma lista](https://blog.alura.com.br/adicionando-elementos-na-lista-do-python-append-ou-extend/), [ordenar uma lista](https://blog.alura.com.br/ordenando-listas-no-python/), [compreensões de lista](https://blog.alura.com.br/simplicando-o-processamento-com-compreensao-de-lista-do-python/) e [outras ferramentas básicas desse tipo](https://blog.alura.com.br/operacoes-basicas-com-listas-no-python).

Se quiser continuar estudando Python, temos alguns [cursos na Alura sobre a linguagem](https://www.alura.com.br/cursos-online-programacao/python) para você seguir em frente com seu aprendizado!