



12

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Texto base

12

Sequências aninhadas e cópias de objetos

Prof. Me. Lucio Nunes de Lira

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Nesta aula os objetivos são: (I) conhecer as sequências aninhadas; (II) usar sequências aninhadas para representar matrizes e tabelas; (III) recordar os conceitos de objetos mutáveis e imutáveis; (IV) entender o mecanismo de cópia de objetos em Python; (V) compreender o conceito e uso de cópias rasas e cópias profundas.

12.1. Motivação

Vimos que sequências são importantes ao lidar com grandes quantidades de dados, como para calcular a média de altura de uma população, a folha de pagamento de uma empresa, contar itens em um estoque e assim por diante, pois podemos usar laços de repetição para iterar sobre essas sequências e realizar operações que na maioria das vezes seriam inviáveis ou impossíveis se dependêssemos apenas de variáveis simples.

No entanto, há problemas que naturalmente apresentam um aninhamento dessas informações, como por exemplo guardar as notas de todos os alunos de todas as turmas de uma instituição de ensino. Aqui temos uma lista de turmas, cada turma é uma lista de alunos e para cada aluno temos uma lista de notas. Isto é, são sequências de sequências!



VOCÊ CONHECE?



Dennis Ritchie foi um cientista da computação, criador da linguagem de programação C e co-criador do sistema operacional Unix.

Em 1983, juntamente com Ken Thompson, recebeu o prêmio Turing por sua contribuição relacionada aos sistemas operacionais. Até hoje, C é referência para outras linguagens de programação como C# e Java.

Fonte da imagem: <http://brainprick.com/dennis-ritchie-the-creator-of-c-and-unix/>

12.2. Sequências aninhadas

Em Python, listas e tuplas são sequências que podem conter itens de qualquer tipo, inclusive outras sequências¹, portanto é comum trabalharmos com listas de listas, tuplas de tuplas, listas de tuplas etc. Para que uma sequência seja aninhada, basta colocá-la como item de outra sequência, como mostrado na Codificação 12.1.

```
>>> s = [('pt', 'arroz'), ('en', 'rice'), ('fr', 'riz')]
```

Codificação 12.1: Exemplo de uma lista de tuplas.

O aninhamento pode ser feito com objetos literais, como na Codificação 12.1, ou com objetos que sejam referenciados por variáveis, como mostra a Codificação 12.2.

```
>>> t1 = [10, 11, 12]
>>> t2 = [20, 21, 22]
>>> t3 = [30, 31, 32]
>>> t = [t1, t2, t3]
```

Codificação 12.2: Exemplo de uma lista de listas.

Sabemos que ao indexar uma sequência acessamos um de seus itens. Portanto, para acessar um item de uma sequência aninhada, basta encadear índices, como visto na Codificação 12.3. As funções e métodos vistos até agora continuam válidos para as sequências aninhadas da mesma forma como eram para sequências não aninhadas.

```
>>> t[0]          # 1º item da sequência t é uma lista.
[10, 11, 12]
>>> t[0][2]      # 3º item do 1º item da sequência t é um inteiro.
12
```

Codificação 12.3: Acesso a um item de uma sequência aninhada.

12.3. Representação de matrizes e tabelas

Matrizes e tabelas bidimensionais podem ser representadas como sequências aninhadas. Matrizes geralmente são numéricas e formadas por itens dispostos em linhas e colunas, para representá-las em Python precisamos definir a sequência principal e as sequências aninhadas, que representam as linhas da matriz e seus itens as colunas. Tabelas são semelhantes às matrizes, porém não precisam ser numéricas ou com itens do mesmo tipo, suas linhas são chamadas de registros e cada coluna da linha é um campo daquele registro. Note que a diferença entre as duas estruturas é conceitual, ambas são compostas por linhas e colunas, e são representadas em Python de forma semelhante.

¹ Neste capítulo, se não houver indicação em contrário, o termo “sequência” será usado para referenciar apenas listas e tuplas, pois intervalos e *strings* são sequências homogêneas e não são aptas a conterem itens que sejam outras sequências, por mais que possam ser itens de outras sequências. A *string* é uma sequência exclusivamente de caracteres e o intervalo uma sequência exclusivamente de números inteiros.

Na Figura 12.1 há um exemplo de matriz que contém as notas de 4 alunos em 3 atividades. Em Python, podemos criar uma variável para cada aluno-nota (12 variáveis), ou uma lista para cada aluno (4 variáveis), como na Codificação 12.4 e Codificação 12.5, respectivamente. Por concisão, utilizamos atribuição paralela na Codificação 12.4.

	nota 0	nota 1	nota 2		
aluno 0	5.5	7.0	8.7	← linha 0	Matriz numérica 4x3 (4 linhas por 3 colunas)
aluno 1	8.0	6.0	9.2	← linha 1	
aluno 2	7.8	8.3	8.5	← linha 2	
aluno 3	0.0	9.9	9.1	← linha 3	
	↑ coluna 0	↑ coluna 1	↑ coluna 2		

Figura 12.1: Exemplo de matriz numérica. Fonte: Elaborado pelo autor.

```
aluno_0_0, aluno_0_1, aluno_0_2 = 5.5, 7.0, 8.7
aluno_1_0, aluno_1_1, aluno_1_2 = 8.0, 6.0, 9.2
aluno_2_0, aluno_2_1, aluno_2_2 = 7.8, 8.3, 8.5
aluno_3_0, aluno_3_1, aluno_3_2 = 0.0, 9.9, 9.1
```

Codificação 12.4: Representação da matriz da Figura 12.1 com variáveis simples.

```
aluno_0 = [5.5, 7.0, 8.7]
aluno_1 = [8.0, 6.0, 9.2]
aluno_2 = [7.8, 8.3, 8.5]
aluno_3 = [0.0, 9.9, 9.1]
```

Codificação 12.5: Representação da matriz da Figura 12.1 com listas simples.

Com a representação da Codificação 12.5, poderíamos calcular a média dos alunos com quatro chamadas à função `calcula_media` da Codificação 12.6, passando uma lista por vez como argumento, conforme a Codificação 12.7.

```
def calcula_media(aluno):
    soma = 0
    for nota in aluno:
        soma += nota
    return soma / len(aluno)
```

Codificação 12.6: Função para cálculo da média dos valores de uma sequência.


```
media_0 = calcula_media(aluno_0)
media_1 = calcula_media(aluno_1)
media_2 = calcula_media(aluno_2)
media_3 = calcula_media(aluno_3)
```

Codificação 12.7: Cálculo da média dos quatro alunos da Figura 12.1.

É possível representar a matriz da Figura 12.1 de forma mais adequada, sem usar 12 variáveis simples ou 4 listas. Até porque, caso a quantidade de alunos ou atividades aumentasse, usar variáveis simples ou mesmo listas simples poderia se tornar inviável. Por isso, criaremos uma matriz usando listas aninhadas, conforme Codificação 12.8.

```
alunos = [[5.5, 7.0, 8.7],
          [8.0, 6.0, 9.2],
          [7.8, 8.3, 8.5],
          [0.0, 9.9, 9.1]]
```

Codificação 12.8: Representação de uma matriz com listas aninhadas.

Na Codificação 12.8 foi criada uma matriz de notas de alunos, em que cada linha representa um aluno e cada coluna representa uma nota do respectivo aluno. Veja que chamamos a matriz de `alunos`, pois essa variável referencia uma lista em que cada item é uma lista de notas de um aluno.

Para criar uma tabela usando sequências aninhadas, como a ilustrada na Figura 12.2, o procedimento é semelhante e a pode ser visto na Codificação 12.9.

Tabela com 3 registros de 4 campos
(3 linhas por 4 colunas)

	tipo	estoque	preço	importado	
produto 0	smartphone	100	1199.00	sim	← registro 0
produto 1	televisão	5	2599.00	não	← registro 1
produto 2	notebook	20	4500.00	sim	← registro 2
	↑ campo 0	↑ campo 1	↑ campo 2	↑ campo 3	

Figura 12.2: Exemplo de tabela. Fonte: Elaborado pelo autor.

```
produtos = [['smartphone', 100, 1199.00, True ],
            ['televisão', 5, 2599.00, False],
            ['notebook', 20, 4500.00, True ]]
```

Codificação 12.9: Representação de uma tabela com listas aninhadas.

12.4. Percorrer matrizes e tabelas

Podemos usar um laço para percorrer as linhas de uma matriz/tabela (que são listas aninhadas) e para cada linha usar um laço para percorrer suas colunas (que são itens da lista aninhada). Assim, podemos, por exemplo, calcular todas as médias da matriz da Codificação 12.8 com apenas dois laços, conforme a Codificação 12.10.

```
medias = []
for aluno in alunos:
    soma = 0
    for nota in aluno:
        soma += nota
    media = soma / len(aluno)
    medias.append(media)
```

Codificação 12.10: Cálculo da média em uma lista aninhada.

Desta forma, mesmo que `alunos` contenha milhares de alunos, podemos calcular a média sem alterar a Codificação 12.10. Mas podemos melhorar! Construiremos uma função que receberá como argumento uma matriz de notas de alunos de uma turma e retornará uma lista com as médias dos alunos. Assim, será possível usar a mesma função para o cálculo das médias de diversas turmas. Veja como na Codificação 12.11.

```
def calcula_medias(turma):
    medias = []
    for aluno in turma:
        media = calcula_media(aluno)
        medias.append(media)
    return medias
```

Codificação 12.11: Função para cálculo da média de todos os alunos de uma turma.

Note que na Codificação 12.11 usamos a função `calcula_media` para calcular a média de cada aluno, e por conta disso não há laços aninhados explicitamente. Essa estratégia simplifica a codificação e é uma das vantagens de usar funções, mas é importante observar que Python ainda executará um laço aninhado, só que implícito.

Quando criamos uma sequência, é comum nomeá-la com o plural de seus itens, mas às vezes o nome da sequência torna-se mais significativo ao se considerar o contexto. Por exemplo, uma sequência de notas pode se tornar “aluno”, pois são as notas de um aluno, uma sequência de alunos pode se tornar “turma”, pois são os alunos de uma turma. Não existe regra, use a forma que tornar a solução mais clara, coerente e legível.

Como já discutido, é uma abordagem comum em Python evitar a alteração dos argumentos recebidos pelas funções, porém isso depende do problema a ser resolvido, pois a solução se adequa ao problema e não o problema à solução. Para exemplificação,

leia o enunciado: “crie uma função que receba como argumento uma matriz de notas em que cada linha representa as notas de um aluno em escala zero a dez. A função deve atualizar a matriz para a escala zero a vinte, mantendo a proporção”. Na Codificação 12.12 há uma solução sem efeitos colaterais, ou seja, não altera o argumento passado.

```
def atualiza_notas_1(turma):
    turma_atualizada = []
    for aluno in turma:
        aluno_atualizado = []
        for nota in aluno:
            aluno_atualizado.append(nota * 2)
        turma_atualizada.append(aluno_atualizado)
    return turma_atualizada
```

Codificação 12.12: Atualização das notas sem modificar o argumento.

Porém, existem situações onde é desnecessário preservar o argumento ou em que o objetivo é exatamente alterá-lo. Nesses casos, a Codificação 12.13 pode ser usada.

```
def atualiza_notas_2(turma):
    for aluno in turma:
        for i in range(len(aluno)):
            aluno[i] *= 2
```

Codificação 12.13: Atualização das notas modificando o argumento.

Observe que não precisamos mudar os itens de `turma`, apenas os itens das sequências aninhadas em `turma`, logo o laço externo não precisa ser alterado, apenas o laço interno, que agora irá operar sobre os índices de cada `aluno`, para que os valores possam ser alterados. Porém, também podemos usar a abordagem da Codificação 12.14, em que os itens da matriz são acessados por meio de seus dois índices: linha e coluna.

```
def atualiza_notas_3(turma):
    for linha in range(len(turma)):
        for coluna in range(len(turma[linha])):
            turma[linha][coluna] *= 2
```

Codificação 12.14: Atualização de notas com acesso à matriz por sua linha e coluna.

12.5. Preencher matrizes e tabelas

É importante saber como preencher matrizes e tabelas com entradas dadas pelo usuário. Na Codificação 12.15 há um programa em que a função `coleta_notas` é responsável por preencher e retornar uma lista de notas de um aluno e a função `preenche_turma` chama diversas vezes a função `coleta_notas` para preencher uma matriz de notas de uma turma com `qtd_alunos` linhas e, ao final, retorna a matriz criada. Há na Figura 12.3 um exemplo de execução. Faça testes no Python Tutor.

```
def coleta_notas():
    notas = input().split()
    for i in range(len(notas)):
        notas[i] = float(notas[i])
    return notas

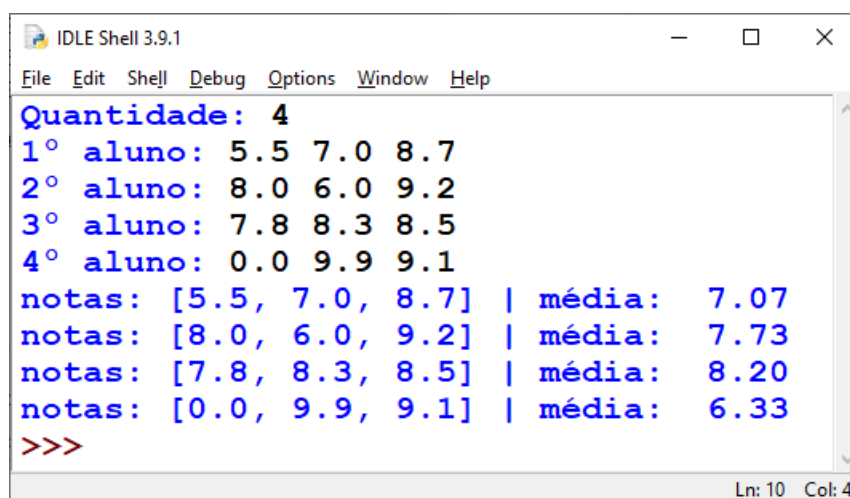
def preenche_turma(qtd_alunos):
    turma = []
    for i in range(qtd_alunos):
        print(f'{i + 1}º aluno:', end=' ')
        aluno = coleta_notas()
        turma.append(aluno)
    return turma

def calcula_media(aluno):
    soma = 0
    for nota in aluno:
        soma += nota
    return soma / len(aluno)

def resumo_turma(turma):
    for aluno in turma:
        media = calcula_media(aluno)
        print(f'notas: {aluno} | média: {media:5.2f}')

qtd_alunos = int(input('Quantidade: '))
turma = preenche_turma(qtd_alunos)
resumo_turma(turma)
```

Codificação 12.15: Programa completo para coleta de notas e exibição de médias.



```
IDLE Shell 3.9.1
File Edit Shell Debug Options Window Help
Quantidade: 4
1º aluno: 5.5 7.0 8.7
2º aluno: 8.0 6.0 9.2
3º aluno: 7.8 8.3 8.5
4º aluno: 0.0 9.9 9.1
notas: [5.5, 7.0, 8.7] | média: 7.07
notas: [8.0, 6.0, 9.2] | média: 7.73
notas: [7.8, 8.3, 8.5] | média: 8.20
notas: [0.0, 9.9, 9.1] | média: 6.33
>>>
```

Figura 12.3: Exemplo de execução da Codificação 12.15. Fonte: Elaborado pelo autor.



VAMOS PRATICAR!

- 1) Crie uma função que receba uma matriz numérica como argumento e retorne o valor da soma de seus itens. Acesse os itens da matriz indexando-a com os dois índices.
- 2) Crie uma função que receba uma matriz numérica como argumento e devolva a matriz oposta, ou seja, uma cópia da matriz argumento com os itens com sinal invertido.



VOCÊ SABIA?

Em Python, uma das principais ferramentas para trabalhar com vetores e matrizes é a biblioteca Numpy, desenvolvida para realizar com maior eficiência cálculos complexos com diferentes tipos de sequências. É uma biblioteca utilizada em áreas como ciência de dados, engenharia, matemática aplicada, estatística, economia, entre outras.

Essa biblioteca, como outras do CPython, foi desenvolvida em C, mas sua interface é Python, possibilitando a eficiência dos códigos em C, com a simplicidade da sintaxe Python. Portanto, escreve-se um código fácil de ler e entender, sem abrir mão do desempenho necessário para aplicações que tratam com enorme quantidade de dados.

O Numpy, juntamente com outras bibliotecas do Python que dependem dele, como Scikit-image, SciPy, Matplotlib e Pandas, foi usado em duas situações com repercussão mundial: a geração da primeira imagem de um buraco negro (Numpy, 2020a) e a detecção de ondas gravitacionais (Numpy, 2020b) pelos cientistas do Observatório de Ondas Gravitacionais por Interferômetro Laser (LIGO).

Em ambos os casos, Python foi usado para coletar, tratar, analisar e gerar visualizações com terabytes de dados diários. São aplicações de Python em problemas complexos, com muitos dados e com exigência de excelente desempenho computacional.

12.6. Cópia de objetos em Python

Em Python a atribuição de um objeto a uma variável não gera uma cópia do objeto, pois a variável receberá apenas uma referência (endereço) ao objeto que já foi criado e está na memória. Veja na Figura 12.4 um exemplo no *Python Tutor* em que uma lista é atribuída a diversas variáveis, o conceito se aplica a qualquer objeto da linguagem.

Note que a lista foi criada apenas uma vez, na primeira atribuição, porém outras variáveis também a referenciam, logo qualquer uma das variáveis poderá ser usada para acessar e alterar o conteúdo dessa lista, e isso será refletido nas demais, pois todas apontam para o mesmo objeto na memória e este objeto é mutável.

Caso seja necessário criar uma cópia do objeto que está na memória para que, por exemplo, a alteração feita na cópia não afete o objeto original, deve-se solicitar uma cópia explicitamente. Aprenderemos como fazer isso, no entanto é útil ressaltar que

Python 3.6
([known limitations](#))

```
1 lista_1 = [1, 2, 3]
2 lista_2 = lista_1
→ 3 lista_3 = lista_2
```

[Edit this code](#)

→ line that just executed
→ next line to execute

Global frame

list

0	1	2
1	2	3

Done running (3 steps)

12.6.1. Cópia rasa

Em Python há diversas formas de conseguirmos uma cópia rasa de objetos mutáveis, que podem variar dependendo do tipo de objeto. A forma genérica, que funciona para todos os objetos mutáveis, é com a função `copy` do módulo homônimo `copy`, como mostra a Figura 12.5.

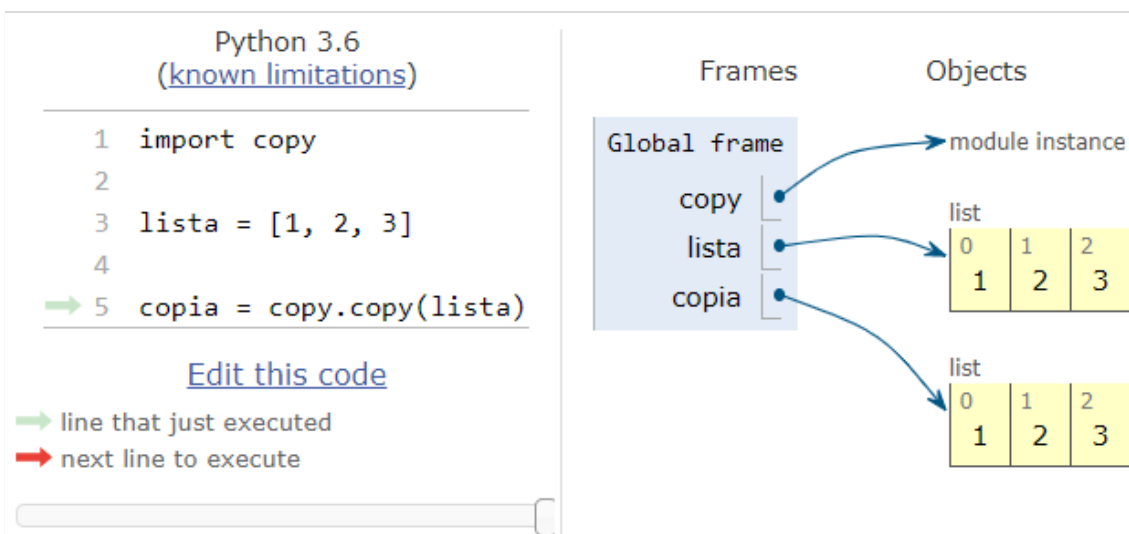


Figura 12.5: Cópia rasa de um objeto mutável com copy. Fonte: Elaborado pelo autor.

Para uma lista `s`, temos ainda as seguintes formas de obtermos cópias rasas:

- Com um fatiamento completo: `s[:]`
- Com o método `copy` dos objetos do tipo lista: `s.copy()`
- Com a função construtora de listas: `list(s)`

Assim como a função `copy`, as três formas anteriores retornam um novo objeto lista, contendo itens com as mesmas referências da original, o que pode ser visto na Figura 12.6. Contudo, esse tipo de cópia, dependendo do objeto copiado, não desvincula a cópia completamente do objeto original, veremos mais detalhes no próximo tópico.

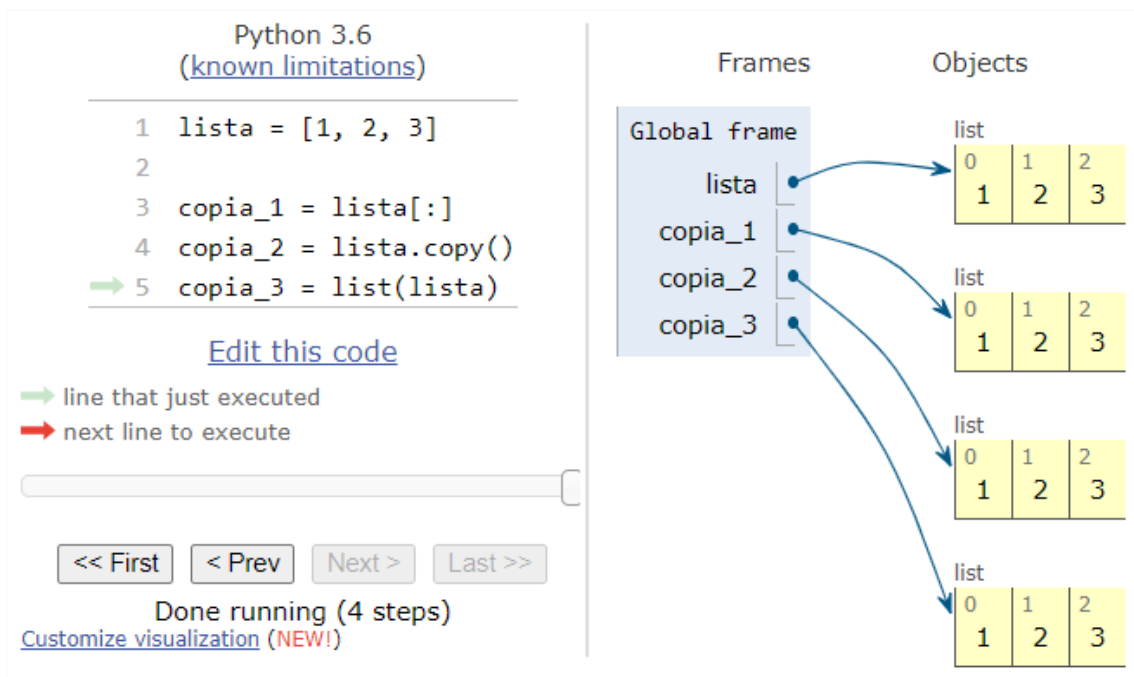


Figura 12.6: Cópias rasas de uma lista. Fonte: Elaborado pelo autor.

12.6.2. Cópia profunda

Ao copiar uma lista que não contém listas aninhadas, uma cópia rasa gera um novo objeto totalmente independente do original, mas veja na Figura 12.7 o que ocorre ao criarmos cópias rasas de uma lista de listas.

As formas que vimos para criar cópias rasas geram apenas uma cópia da lista principal, a mais externa, mas não das listas aninhadas. Desta forma, a cópia da lista original referencia as mesmas listas aninhadas que a lista original!

Portanto, nas situações em que é necessário uma cópia totalmente desvinculada do objeto original, é preciso criar uma cópia profunda, isto é, uma cópia do objeto principal e dos objetos internos a ele em todos os níveis. Veja na Figura 12.8 uma cópia profunda com base no exemplo da Figura 12.7. Em Python, cópias profundas são realizadas com a função `deepcopy` do módulo `copy`.

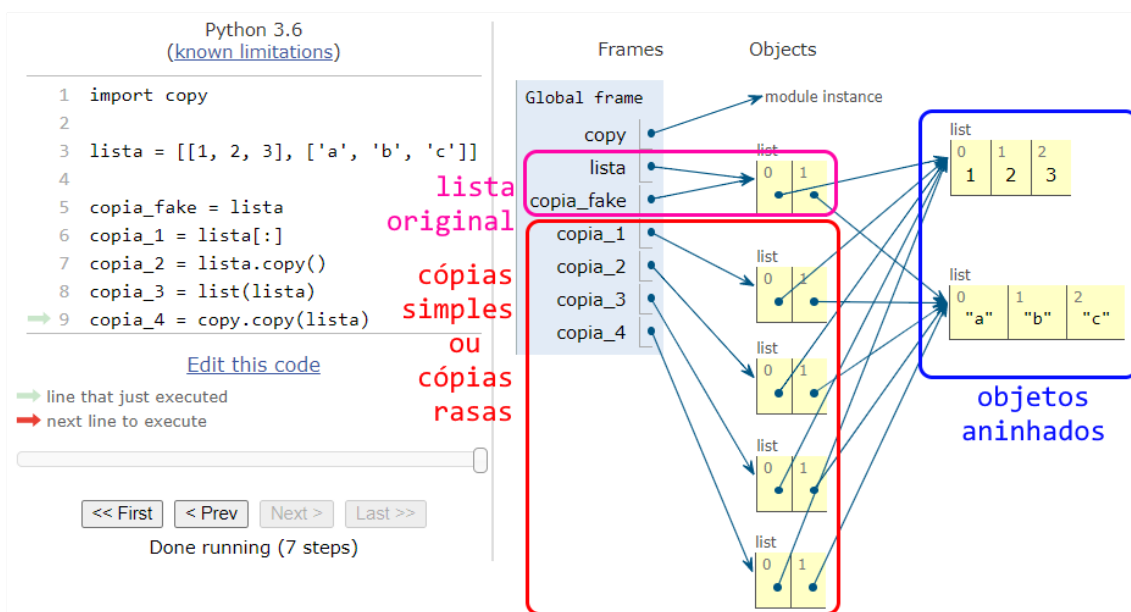


Figura 12.7: Cópias rasas de uma lista de listas. Fonte: Elaborado pelo autor.

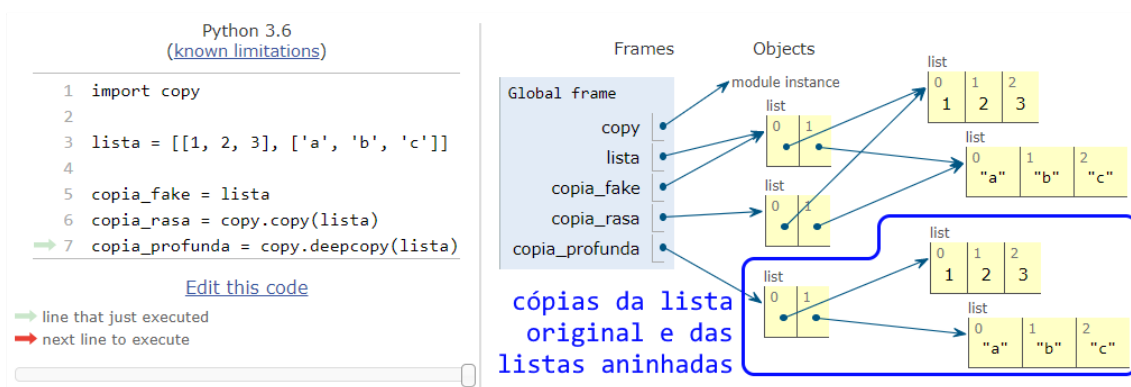


Figura 12.8: Cópia profunda de uma lista de listas. Fonte: Elaborado pelo autor.

12.6.3. Cópia de objetos imutáveis

Como sabemos, é dispensável copiar objetos imutáveis, pois eles não podem ser alterados. Porém, cópias profundas de objetos imutáveis que contenham objetos mutáveis podem ser úteis, como tuplas de listas, porém não abordaremos essa situação aqui.

Veja na Figura 12.10 um exemplo no *Python Tutor* sobre o que ocorre quando atribuímos um novo objeto a uma variável. Para reproduzir este exemplo com objetos simples como números, é preciso alterar a forma padrão com a qual o Python Tutor representa objetos na memória. Ajuste-o conforme a Figura 12.9.

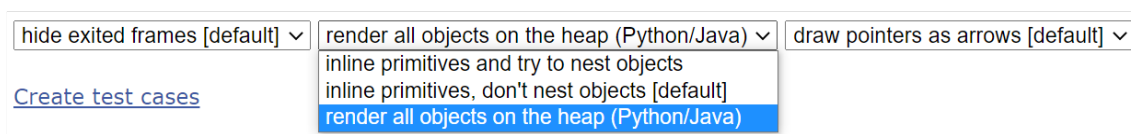


Figura 12.9: Alteração da renderização do Python Tutor. Fonte: Elaborado pelo autor.

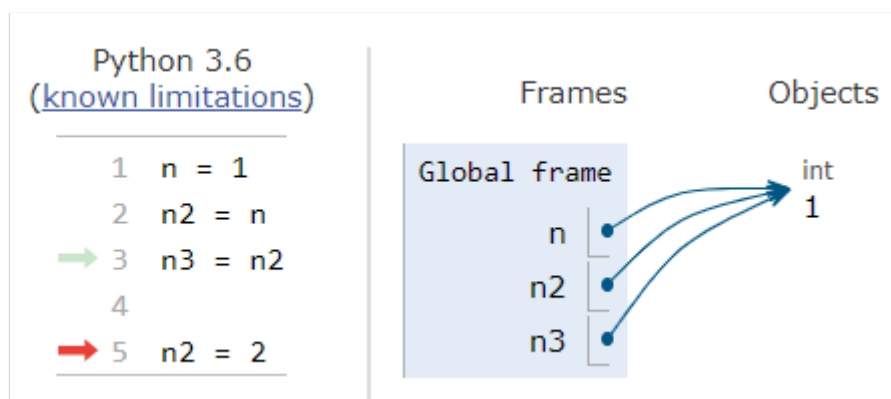


Figura 12.10: Estado da memória após a 3ª instrução. Fonte: Elaborado pelo autor.

Na 1ª instrução, Python cria um objeto na memória do tipo inteiro com o valor **1** e atribui a referência à variável **n**. Na 2ª instrução, a mesma referência é atribuída à variável **n2**, por meio de **n**. Na 3ª instrução, a referência ao mesmo objeto é atribuída à **n3**, por meio de **n2**. Ou seja, três variáveis que referenciam o mesmo objeto.

Na Figura 12.11 vemos que, após a 4ª instrução, o valor de **n2**, que antes era a referência ao objeto **1** na memória, foi sobrescrito pela referência ao objeto **2**, recém criado e atribuído à **n2**. Todo objeto é criado na primeira vez que ocorre no código.

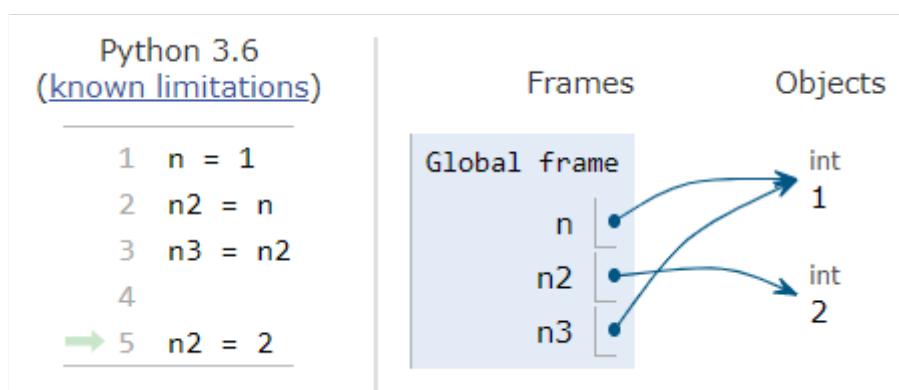


Figura 12.11: Estado da memória após a 4ª instrução. Fonte: Elaborado pelo autor.

Bibliografia e referências

Case Study: First Image of a Black Hole. **Numpy** 2020a. Disponível em: <<https://numpy.org/case-studies/blackhole-image/>>. Acesso em: 20 mar. 2021.

Case Study: Discovery of Gravitational Waves. **Numpy** 2020b. Disponível em: <<https://numpy.org/case-studies/gw-discov/>>. Acesso em: 20 mar. 2021.

DOWNEY, A. B. **Pense em Python**. 1 ed. São Paulo: Novatec Editora Ltda., 2016.

STURTZ, J. Lists and Tuples in Python. **Real Python**, 2018. Disponível em: <<https://realpython.com/python-lists-tuples/#lists-can-be-nested>>. Acesso em: 20 mar. 2021.