9

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

9

Sequências e estrutura de repetição definida (*for*)

Prof. Me. Lucio Nunes de Lira
Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Nesta aula os objetivos são: (I) compreender o que são sequências; (II) conhecer os tipos de sequências em Python: strings, listas, tuplas e intervalos; (III) distinguir sequências mutáveis e imutáveis; (IV) distinguir sequências homogêneas e heterogêneas; (V) percorrer sequências por meio de seus índices; (VI) entender a estrutura de repetição com quantidade de repetições definida: laço for; (VII) percorrer sequências por meio de seus itens com o laço for.

9.1. Motivação

Ao lidarmos com programas maiores e mais complexos, torna-se evidente a necessidade manipular mais dados e, consequentemente, melhorar o gerenciamento deles. Logo, precisamos de estruturas eficientes para manipulação de dados. Neste capítulo introduziremos estruturas que permitirão armazenar grandes quantidades de dados e também um laço de repetição bastante útil para trabalhar com elas!

9.2. Introdução

Não é raro lidarmos com problemas que exigem manipulação de uma grande quantidade de dados. Nestes casos, pensar em armazená-los em variáveis simples é algo difícil, indesejável e, às vezes, impossível. Para essas situações aprenderemos o conceito de sequências, um recurso importante e frequente em programas mais avançados.

No entanto, não é muito útil armazenar uma imensa quantidade de dados se não puderem ser acessados de modo simples e rápido, para isso aprenderemos a segunda estrutura de repetição do Python, o laço de repetição `for`, em que a quantidade de

repetições do bloco de código é definida antes do *loop* ser iniciado. Este laço em Python possui uma sintaxe que o torna especialmente útil para acessar itens de sequências, facilitando a escrita e a leitura do código.

9.3. Sequências

Ao trabalhar com soluções para problemas mais complexos, é frequente a necessidade de manipulação de grandes quantidades de dados. Para lidar com muitos dados, é indispensável uma forma eficiente de armazenamento e acesso, evitando, por exemplo, a criação de diversas variáveis. Até porque, em certos problemas, é impossível criar a quantidade necessária de variáveis para guardar todos os valores, pois muitas vezes a quantidade exata só será conhecida durante a execução do programa. Para esses casos, usaremos um recurso da linguagem Python: as *sequências*.

Podemos imaginar uma sequência como uma variável com *diversos compartimentos*, chamados de *itens da sequência*, inclusive o termo "sequência" indica que há noção de ordem entre os itens (1º, 2º, 3º etc.). Cada item é identificado por dois valores: o nome da variável à qual a sequência foi atribuída e um número inteiro referente à sua posição na sequência, chamado *índice*.

Em Python, os índices são contados a partir do zero, em ordem crescente, logo o primeiro item tem índice zero, o segundo tem índice um, o terceiro tem índice dois e assim sucessivamente.

Denominamos como *tamanho da sequência* a quantidade de itens que ela possui. Note que, como consequência da forma como os índices são associados aos itens, começando em zero, o último índice sempre será um a menos que o tamanho da sequência, ou seja, se tivermos uma sequência com 5 itens, o último terá índice 4. O último índice é, portanto, dado pela fórmula $tamanho\ da\ sequência\ -\ 1$.

No entanto, Python possui uma característica pouco usual em outras linguagens, que é a existência de índices negativos ou regressivos, isto é, os itens são associados a dois índices, um que marca a posição relativa ao início da sequência e outro que marca a posição relativa ao fim dela. Logo, o último item da sequência também está associado ao índice `-1`, o penúltimo ao índice `-2` e assim sucessivamente, sempre reduzindo uma unidade a cada item mais próximo ao início da sequência. A Figura 9.1 é uma ilustração simplificada de uma sequência de números inteiros que foi atribuída à variável `idades`.

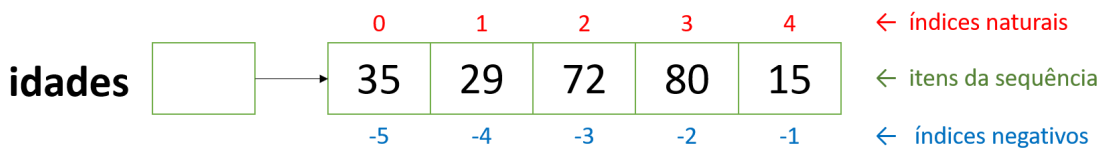

Figura 9.1: Uma sequência atribuída a uma variável. Fonte: Elaborado pelo autor.

Para acessar um item específico da sequência basta *indexá-la* adequadamente com a posição em que está o item desejado. A indexação é feita colocando o índice desejado entre colchetes, após o nome da variável. Por exemplo, utilizando a sequência ilustrada na Figura 9.1, podemos acessar o quarto item, que corresponde ao número `80`, tanto com a instrução `idades[3]` quanto com a instrução `idades[-2]`. A escolha de qual índice usar dependerá do algoritmo, o uso do índice natural é mais comum, mas quando, por algum motivo, for necessário acessar o penúltimo item, é mais fácil usar o índice `-2`, pois torna desnecessário conhecer o tamanho da sequência.

Em Python, há um hábito de que erros não devem ocorrer silenciosamente, pois isso pode acarretar comportamentos inesperados e difíceis de prever, então se uma sequência for indexada com índice inválido, isto é, inexistente na sequência, será gerado um erro de execução.

O Python possui diversas funções e métodos que nos ajudam a trabalhar com sequências, por exemplo a função `len`, que recebe como argumento uma sequência e devolve um número inteiro indicando o tamanho desta sequência. Com base na Figura 9.1, ao executar a instrução `len(idades)`, obtemos como retorno o valor `5`. Neste capítulo não focaremos nessas funções e métodos, porém aprenderemos sobre as características e conceitos associados aos diferentes tipos de sequências do Python.

9.3.1. Sequências mutáveis e imutáveis

Existem sequências *mutáveis* e *imutáveis*. Nas sequências mutáveis cada item se comporta de modo semelhante a uma variável comum, portanto pode receber atribuições, sobrescrevendo seus valores iniciais. Nas sequências imutáveis, uma vez definida a sequência, seus itens não podem mais ser modificados. Veja na Figura 9.2 uma atribuição de valor a um item de uma sequência mutável e sua consequente mudança de estado em decorrência desta operação.

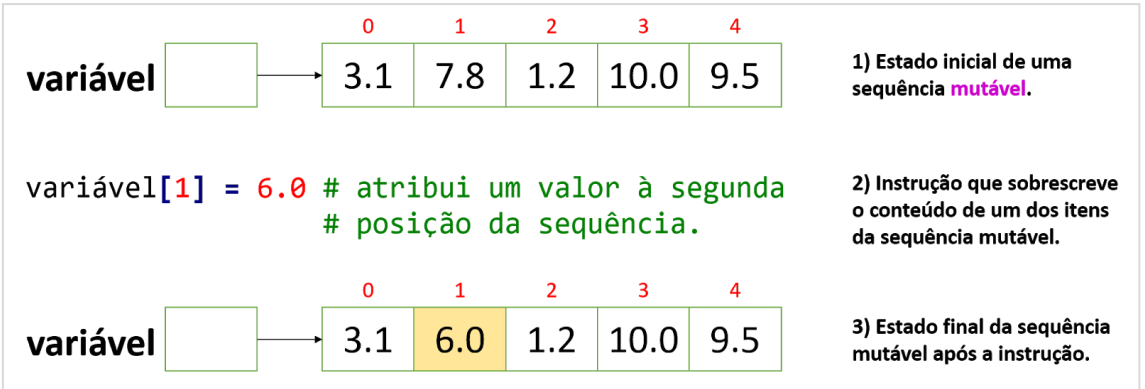

Figura 9.2: Modificação de item de uma sequência mutável. Fonte: Elaborado pelo autor.

Na Figura 9.3, há uma tentativa de atribuição de valor a um item de uma sequência imutável, o que irá gerar um erro por ser uma operação inválida.

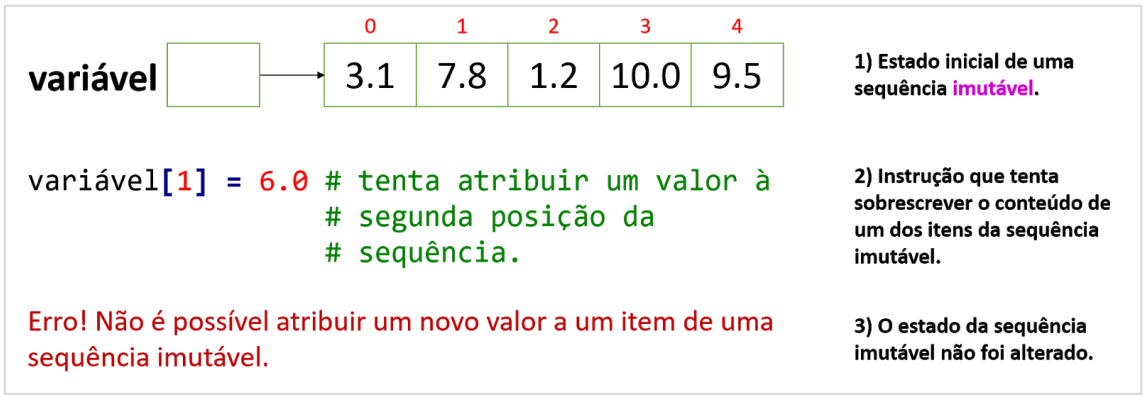

Figura 9.3: Tentativa de modificação de item de uma sequência imutável.
Fonte: Elaborado pelo autor.

Note que itens de sequências imutáveis não podem ser modificados, porém isso não impede que a variável que referencia a sequência receba um novo valor. É importante observar que, desta forma, sobrescreve-se o valor da variável e não o valor de um item da sequência. A Figura 9.4 ilustra uma atribuição a uma variável que referencia uma sequência (mutável ou imutável, é irrelevante) e, consequentemente, deixa de referenciá-la.

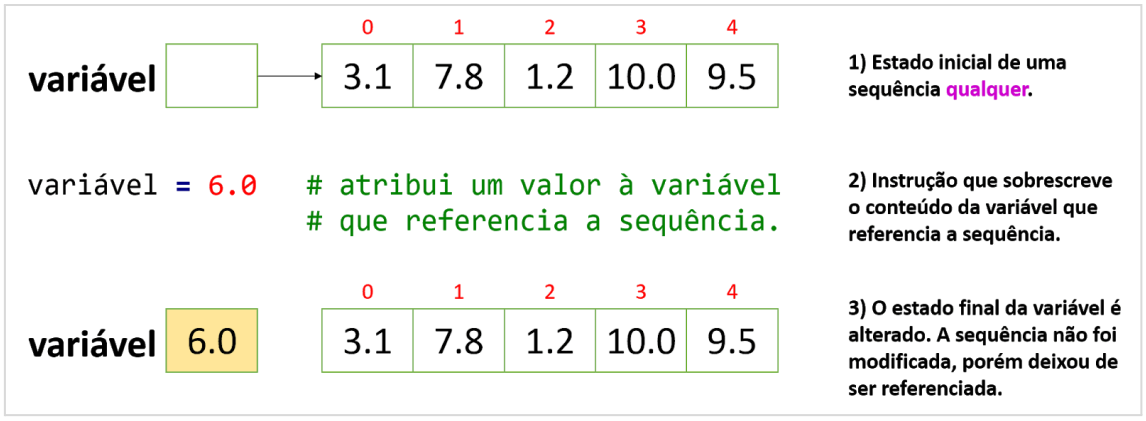

Figura 9.4: Modificação da variável que referencia uma sequência.
Fonte: Elaborado pelo autor.

No exemplo da Figura 9.4, caso a sequência não esteja referenciada por outra variável qualquer, será automaticamente apagada da memória pelo Python, pois será interpretado que é uma sequência sem uso, pois não há como acessá-la. A relação completa de operações disponíveis para sequências é chamada de "Operações Comuns de Sequências" (PSF[1], 2021a). Para as sequências mutáveis, também existem as operações listadas em "Tipos de Sequências Mutáveis" (PSF, 2021b).

[1] PSF - *Python Software Foundation.*

9.3.2. Sequências homogêneas e heterogêneas

Existem sequências *homogêneas* e *heterogêneas*, característica que indica a flexibilidade em armazenar itens de tipos distintos simultaneamente. Em sequências homogêneas, todos os itens devem ser do mesmo tipo, portanto, caso um item seja um número inteiro, todos os demais deverão ser números inteiros, caso seja um caractere, todos os demais também serão caracteres, e assim por diante. Em sequências heterogêneas os itens podem ser de tipos distintos, logo uma mesma sequência pode conter itens *float*, *int*, *bool* etc. A Figura 9.5 ilustra esse conceito.

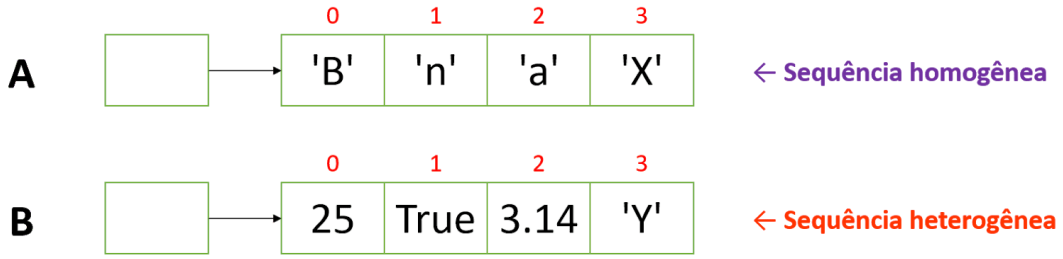

Figura 9.5: Exemplos de sequências homogênea e heterogênea.
Fonte: Elaborado pelo autor.

9.3.3. Tipos de sequências

O Python possui três tipos básicos de sequências: listas (*list*), tuplas (*tuple*) e intervalos (*range*). Também há um tipo de sequência específica para armazenar caracteres (*strings*) e outros tipos de sequência dedicadas para manipulação de dados binários. Nesta aula abordaremos os tipos *string*, lista, tupla e intervalo.

9.3.3.1. *Strings*

Já introduzimos o tipo de dados *string*, porém, propositalmente, não trabalhamos com suas características de sequência. As *strings* em Python são sequências imutáveis e homogêneas, em que todos os itens devem ser caracteres Unicode. Assim, como em outras sequências, é possível acessar um item específico da *string* indexando-a. Podem ser definidas de várias formas:

1) Pares de apóstrofos: `'a casa amarela'`;
2) Pares de aspas: `"a casa amarela"`;
3) Pares de três apóstrofos ou aspas, permitindo que tenham múltiplas linhas: `'''a casa amarela'''` ou `"""a casa amarela"""`;
4) Com o construtor de tipo `str()`, que permite também a conversão de outros tipos de dados para *string*.

Indica-se *string* vazia com um par de delimitadores sem nenhum conteúdo entre eles `''`. Observe que a *string* `' '` não está vazia, pois contém um caractere de espaço. Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.6.

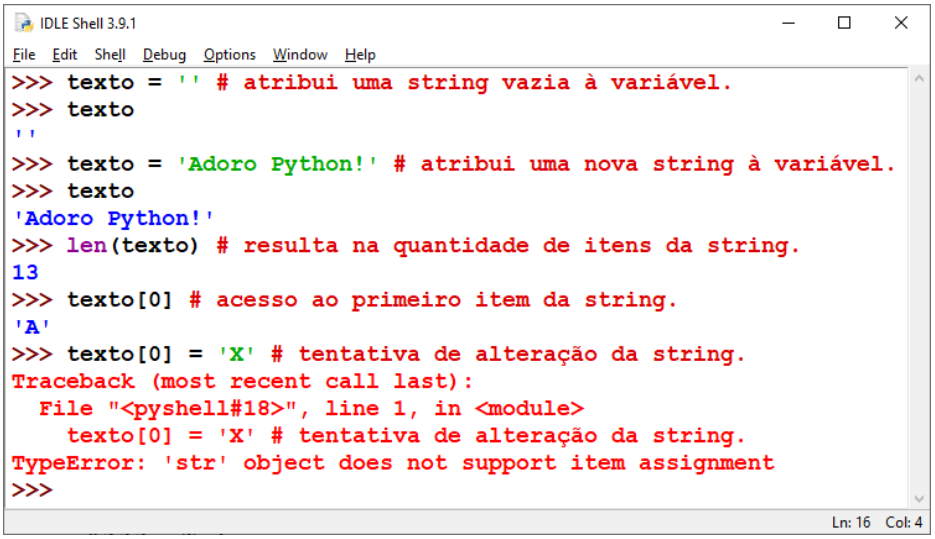

Figura 9.6: Exemplos de operações com *string*. Fonte: Elaborado pelo autor.

9.3.3.2. Listas

Sequências do tipo *list* são mutáveis e heterogêneas, por mais que seja comum usá-las para armazenar itens do mesmo tipo. Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.7, que pode ser definidas de várias formas:

1) Apenas um par de colchetes, para indicar lista vazia: `[]`;
2) Itens separados por vírgulas entre um par de colchetes: `[4, True, 1.5]`;
3) Com compreensão de lista (não abordaremos agora);
4) Com o construtor de tipo `list()`, que pode ser usado também para converter outros tipos de sequências em listas.

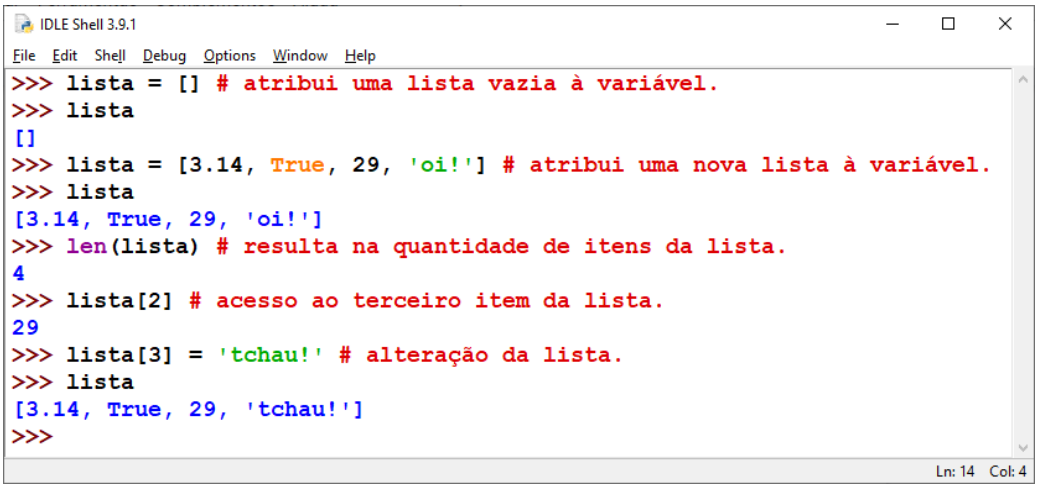

Figura 9.7: Exemplos de operações com lista. Fonte: Elaborado pelo autor.

9.3.3.3. Tuplas

Sequências do tipo *tuple* são imutáveis e heterogêneas, seu uso é comum em situações em que é necessário garantir que a sequência não seja modificada, como em chaves de dicionários ou ao ser passada como argumento para funções que produzam efeitos colaterais. Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.8. Assim como listas, tuplas podem ser definidas de várias formas:

1) Apenas um par de parênteses para indicar tupla vazia: `()`;
2) Uma vírgula à direita, indicando um tupla de só um item: `7,` ou `(7,)`;
3) Itens separados por vírgulas: `4, True, 1.5` ou `(4, True, 1.5)`;
4) Com o construtor de tipo `tuple()`, que pode ser usado também para converter outros tipos de sequências em tuplas.

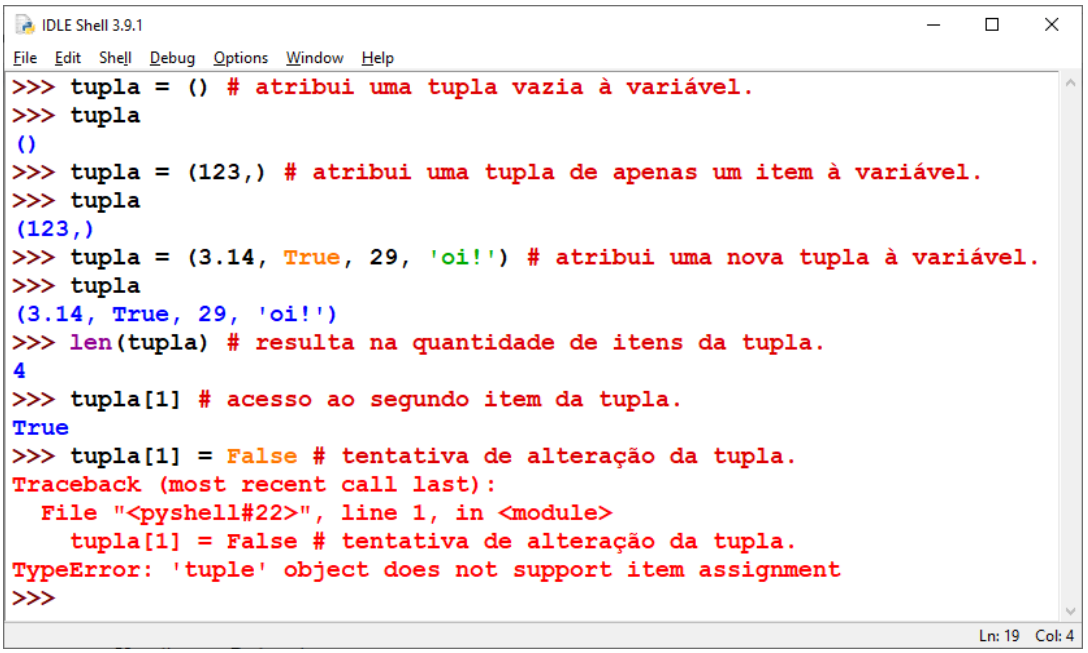

Figura 9.8: Exemplos de operações com tupla. Fonte: Elaborado pelo autor.

9.3.3.4. Intervalos

Sequências do tipo *range* são imutáveis e homogêneas, gerando um intervalo de números inteiros, algo útil principalmente para controle de laços de repetição. Intervalos são criados com uso do construtor `range()` acompanhado dos argumentos `início`, `fim` e `passo`, que devem ser números inteiros. Existem três variações para criação de intervalos, dependendo do número de argumentos:

1) Três argumentos: `range(início, fim, passo)`, gera uma sequência com itens no intervalo `[início..fim[`, variando de `passo` em `passo`;
2) Dois argumentos: `range(início, fim)`, gera uma sequência crescente com itens no intervalo `[início..fim[`, variando de `1` em `1`;

3) Um argumento: `range(fim)`: gera uma sequência crescente com itens no intervalo `[0..fim[`, variando de `1` em `1`.

Observe algumas implicações da forma como sequências *range* funcionam:

a) O intervalo é aberto à direita, portanto `fim` não pertence à sequência;
b) Dependendo de como o construtor é especificado, pode-se gerar um intervalo vazio, por exemplo em `range(10, 10)`, pois não existem inteiros no intervalo `[10..10[`. Ficou em dúvida? Isso equivale a solicitar um intervalo que comece em 10 e termine antes de 10;
c) No construtor com três argumentos, `range(início, fim, passo)`, se `passo` for um número negativo, a sequência será decrescente, desde que `início` seja maior do que `fim`, caso contrário será gerado um intervalo vazio. Logicamente, `passo` zero é inválido;
d) Podemos interpretar `range` como uma função que retorna um intervalo com base em uma progressão aritmética cuja razão entre os termos é o `passo`.

Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.9.

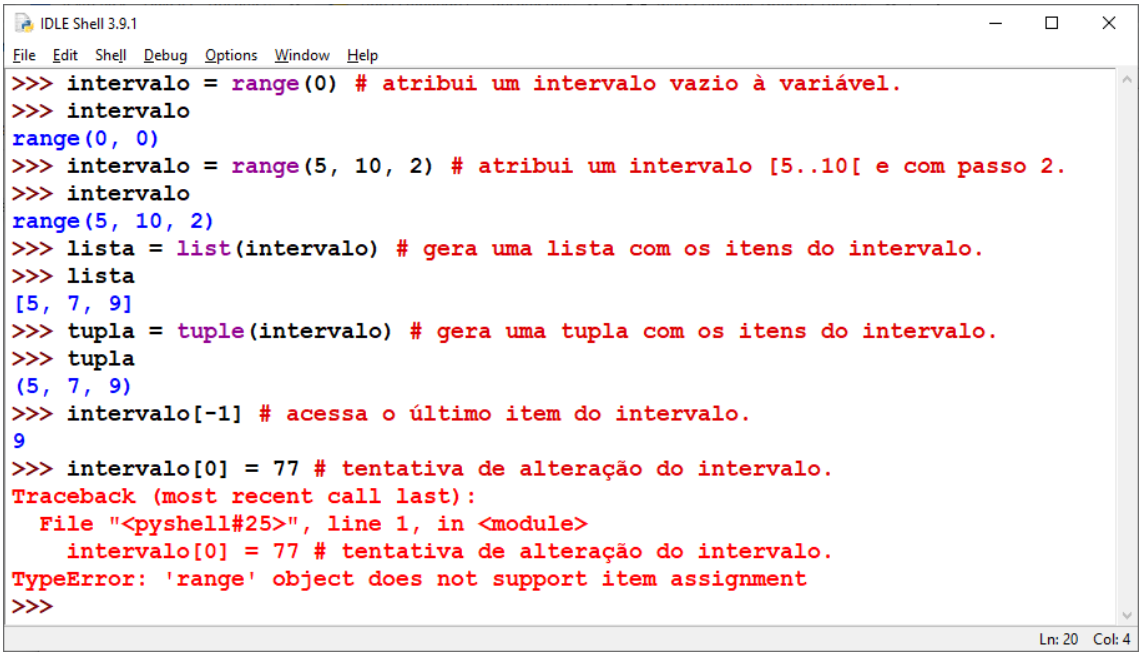

Figura 9.9: Exemplos de operações com intervalo. Fonte: Elaborado pelo autor.

VAMOS LER!

Em Python 3, a sequência *range* é baseada em uma técnica de programação denominada "avaliação preguiçosa", em que o processamento para gerar um valor é propositalmente atrasado até o instante em que o valor seja realmente necessário. Essa estratégia permite economia de recursos computacionais, como processamento e memória. Veja mais detalhes em: Avaliação preguiçosa – Wikipédia, a enciclopédia livre

9.3.4. Percorrendo sequências

Percorrer uma sequência, iterar sobre ela ou "varrê-la", equivale a acessar sistematicamente diversos de seus itens seguindo alguma ordem. Frequentemente, percorre-se uma sequência para acessar seus itens visando exibição, alteração ou utilização dos valores por razões diversas, como para construção de outras sequências. Percursos em sequências costumam ser feitos com uso de laços de repetição.

Para entender a necessidade das sequências e a razão de percorrê-las, pensaremos sobre o seguinte enunciado: *"Crie um programa que receba como entrada quatro salários, calcule a média salarial e exiba os salários abaixo da média"*. A Codificação 9.1 contém uma solução válida para esse problema, porém sem o uso de sequências.

```
soma = 0

salario_0 = float(input('Salário: R$ '))
soma += salario_0

salario_1 = float(input('Salário: R$ '))
soma += salario_1

salario_2 = float(input('Salário: R$ '))
soma += salario_2

salario_3 = float(input('Salário: R$ '))
soma += salario_3

media = soma / 4

if salario_0 < media:
    print(f'Abaixo da média: R$ {salario_0:.2f}')
if salario_1 < media:
    print(f'Abaixo da média: R$ {salario_1:.2f}')
if salario_2 < media:
    print(f'Abaixo da média: R$ {salario_2:.2f}')
if salario_3 < media:
    print(f'Abaixo da média: R$ {salario_3:.2f}')
```

Codificação 9.1: Programa que exibe os salários abaixo da média (sem sequências).

Na Codificação 9.1 foi necessário criar quatro variáveis para guardar quatro entradas do usuário. Se fossem mil salários, seriam necessárias mil variáveis. Porém, podemos construir uma versão alternativa que, ao invés de usar variáveis simples, usará uma lista para guardar os salários, conforme a Codificação 9.2.

```
salarios = [0, 0, 0, 0]
soma = 0

salarios[0] = float(input('Salário: R$ '))
soma += salarios[0]

salarios[1] = float(input('Salário: R$ '))
soma += salarios[1]

salarios[2] = float(input('Salário: R$ '))
soma += salarios[2]

salarios[3] = float(input('Salário: R$ '))
soma += salarios[3]

media = soma / 4

if salarios[0] < media:
    print(f'Abaixo da média: R$ {salarios[0]:.2f}')
if salarios[1] < media:
    print(f'Abaixo da média: R$ {salarios[1]:.2f}')
if salarios[2] < media:
    print(f'Abaixo da média: R$ {salarios[2]:.2f}')
if salarios[3] < media:
    print(f'Abaixo da média: R$ {salarios[3]:.2f}')
```

Codificação 9.2: Programa que exibe os salários abaixo da média (com uso de lista).

Você notou que a Codificação 9.2 é maior que a Codificação 9.1 e talvez imagine que não houve benefício em substituir as variáveis simples pela lista. E você está certo! Do modo como a lista foi usada, não houve ganho e a legibilidade foi prejudicada, isso porque a usamos de modo idêntico às variáveis simples, escrevendo uma instrução para cada acesso e atribuição, mesmo que essas instruções estejam praticamente idênticas, variando apenas os índices dos itens. O que precisamos é de uma abordagem diferente.

Podemos percorrer listas variando apenas seu índice, uma vez que o nome da variável que referencia a sequência é sempre o mesmo. Com esse recurso, podemos usar uma variável como índice e alterá-la de acordo com a necessidade de indexação. Essa flexibilidade permite a abordagem da Codificação 9.3, em que repetimos as atribuições e acessos aos itens usando *loop* ao invés de reescrever as instruções.

```
salarios = [0, 0, 0, 0]
soma = 0

i = 0  # variável que será usada como índice.
while i < 4:
    salarios[i] = float(input('Salário: R$ '))
    soma += salarios[i]
    i += 1

media = soma / 4

i = 0  # variável que será usada como índice.
while i < 4:
    if salarios[i] < media:
        print(f'Abaixo da média: R$ {salarios[i]:.2f}')
    i += 1
```

Codificação 9.3: Programa que exibe os salários abaixo da média (com `while`).

Assim, reduziu-se o código e, principalmente, o programa tornou-se mais flexível, pois caso seja necessário armazenar mais salários, basta realizar pequenas mudanças na codificação, algo potencialmente inviável se estivéssemos limitados às variáveis simples ou sem laços para percorrer sequências.

Agora nossos programas podem trabalhar com grandes quantidades de dados com o uso de sequências e laços de repetição para percorrê-las. No entanto, a linguagem Python possui um segundo *loop* que pode tornar nossos códigos ainda mais concisos e elegantes, principalmente ao lidarmos com sequências, é a estrutura de repetição `for`.

9.4. Estrutura de repetição *for*

Python tem uma estrutura de repetição com quantidade de repetições definida, o `for`. Geralmente, esse laço é usado quando sabemos antecipadamente a quantidade de vezes que o bloco de código deve ser repetido, ou para percorrer sequências.

Em muitas linguagens, `for` é "atalho" para um `while` com número pré-definido de iterações, agrupando em uma mesma linha as três partes que controlam o laço: (I) inicialização da variável de controle; (II) condição de repetição e (III) incremento da variável de controle. Na Codificação 9.4 há um exemplo deste tipo de laço `for`.

```
for (i = 0; i < n; i += 1)
    <bloco de código>
```

Codificação 9.4: Laço `for` tradicional, comum em outras linguagens de programação.

No entanto, esse tipo de laço `for` tradicional não existe em Python. Em seu lugar temos um laço que opera de maneira mais intuitiva sobre as sequências, e que reduz a chance de erros na manipulação dos índices, que é necessária para escrevermos tanto o laço `while`, vistos nos exemplos anteriores, quanto o laço `for` tradicional.

Esse laço é conhecido, em inglês, como `for each`, que pode ser traduzido como "*para cada*", e podemos então interpretar essa estrutura de repetição como "*para cada item da sequência, faça…*". Veja na Codificação 9.5 a sintaxe adotada pelo Python.

```
for <variável> in <sequência>:
    <bloco de código>
```

Codificação 9.5: Estrutura do laço de repetição `for` do Python.

Simplificadamente, podemos entender `<sequência>` como uma sequência de qualquer tipo e tamanho, até vazia. Já `<variável>` é uma variável de controle que, caso não exista previamente, será criada no início da execução do laço, se a sequência não for vazia, pois o primeiro item será atribuído a ela dando início à primeira execução do `<bloco de código>`. Caso exista variável com identificador igual à `<variável>`, seu valor será sobrescrito pela atribuição do primeiro item de `<sequência>`.

O `for` executa seu bloco de código uma vez para cada item de `<sequência>` atribuído à `<variável>`, de modo que o número total de rodadas é facilmente previsto, pois é definido pelo tamanho da sequência. Não é recomendado alterar `<sequência>` durante a execução do `for`, pois pode gerar efeitos inesperados. A Figura 9.10 ilustra o comportamento deste laço.

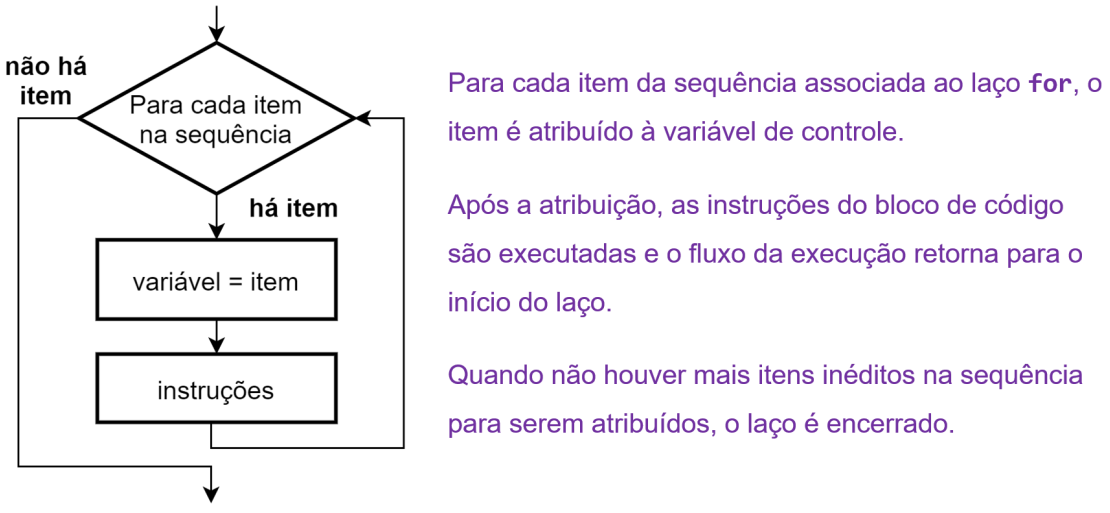

Figura 9.10: Representação em fluxograma da estrutura de repetição `for`.
Fonte: Elaborado pelo autor.

Reescreveremos a Codificação 9.3 usando `for`. Usaremos também um método muito requisitado quando trabalhamos com listas: o `append()`, que anexa ao final da

própria lista um novo item. Assim, é desnecessário que a lista inicie já com todas as posições necessárias[2], deixando o Python gerenciar a alocação de memória da forma que ele julgar mais eficiente. Veja na Codificação 9.6 uma possível solução com `for`.

```
salarios = []
soma = 0

for _ in range(4):
    salario = float(input('Salário: R$ '))
    soma += salario
    salarios.append(salario)

media = soma / 4

for salario in salarios:
    if salario < media:
        print(f'Abaixo da média: R$ {salario:.2f}')
```

Codificação 9.6: Programa que exibe os salários abaixo da média (com `for`).

Na Codificação 9.6 foram usados dois laços `for`: (I) um para receber os valores dos salários, acumulá-los na variável `soma` e adicioná-los à lista e; (II) outro para percorrer a lista de salários e exibir os inferiores a média. Note que o nome da variável referencia seu conteúdo e está no plural, indicando que é uma sequência de salários.

No primeiro laço, usamos como identificador apenas um sublinhado (`_`), o que pode parecer estranho a princípio, porém indica ao leitor do código que não há interesse nos valores assumidos por essa variável. Poderíamos ter usado qualquer outro nome válido, mas esse identificador é comum quando o objetivo do laço é apenas ser executado um número fixo de vezes, sem pretensão de usar os valores da sequência durante a execução do laço.

Note que a função `range` retorna um intervalo `[0..4[`, e portanto o laço será executado 4 vezes, como usamos `append` para incluir os itens na lista, não precisamos nos preocupar em gerenciar índices.

No segundo laço, percorremos a lista de salários, independente do seu tamanho, e os valores dos itens serão usados para algo, no caso uma exibição condicionada a uma verificação. Portanto, foi adotado um identificador que faz referência ao conteúdo da lista, só que agora no singular, pois está representando apenas um item da lista por vez.

[2] Essa inicialização é necessária em linguagens de nível mais baixo, isto é, mais próximas à linguagem de máquina, como C e C++. Esse gerenciamento de alocação de memória é mais trabalho e mais propenso a erros, porém torna o código mais eficiente quando bem feito, sendo útil em diversas aplicações críticas.

Essas convenções adotadas na Codificação 9.6 não são obrigatórias, mas facilitam a leitura do código ao dar maior expressividade às instruções, tornando nosso trabalho como programadores mais fácil e agradável.

VAMOS PRATICAR!

Refaça a solução da Codificação 9.6 para dois novos cenários:

1) O programa perguntará quantos salários serão inseridos, em seguida receberá cada salário, calculará a média e exibirá todos os salários que sejam inferiores à ela.

2) O programa receberá os salários indefinidamente, até que o usuário digite o valor `-1` como salário. Então o programa deve seguir o restante do fluxo, calcular a média e exibir todos os salários inferiores, como no cenário 1. Neste caso, será necessário usar `while`, pois a quantidade de salários não está definida antes da execução do laço.

9.4.1. Exemplos de aplicação do laço *for*

Para melhorar a compreensão sobre o laço de repetiçao `for`, teste os programas das Codificações 9.7 a 9.10.

```
# Crie um programa que exiba o alfabeto minúsculo e maiúsculo.

for letra in 'abcdefghijklmnopqrstuvwxyz':
    print(letra)

print()

for letra in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
    print(letra)
```

Codificação 9.7: Programa que exibe as letras do alfabeto (com `for`).

```
# Crie um programa que exiba, em ordem crescente, os pares de 10
# até 100 e, em ordem decrescente, os ímpares de 100 até 10.

for par in range(10, 101, 2):
    print(par, end=' ')

print()

for impar in range(99, 10, -2):
    print(impar, end=' ')
```

Codificação 9.8: Programa que exibe os pares e ímpares de um intervalo (com `for`).

```
# Crie um programa que exiba os sete dias da semana.

dias = ('domingo', 'segunda', 'terça', 'quarta', 'quinta',
        'sexta', 'sábado')
for dia in dias:
    print(dia)
```

Codificação 9.9: Programa que exibe os setes dias da semana (com `for`).

```
# Crie um programa que leia cinco nomes e exiba a quantidade de
# nomes que começam com vogal.

nomes = []
for _ in range(5):
    nomes.append(input('Nome: '))

qtd = 0
for nome in nomes:
    if (nome[0]=='A' or nome[0]=='E' or nome[0]=='I' or
        nome[0]=='O' or nome[0]=='U'):
            qtd += 1

print(f'{qtd} dos nomes começam com vogal')
```

Codificação 9.10: Programa que conta nomes que começam com vogal (com `for`).

VOCÊ SABIA?

Esse tipo de laço `for` é tão útil e vantajoso que diversas linguagens de programação têm uma implementação equivalente, como C, C++, C#, Java, JavaScript, Pascal, Perl, PHP, Ruby, Rust e Visual Basic. Cada linguagem adota uma sintaxe, mas todas com o mesmo propósito: percorrer um conjunto de itens, um de cada vez, sem a necessidade de manter um índice explícito, executar um bloco de instruções para cada item e encerrar quando não houver mais itens. Veja mais em: https://en.wikipedia.org/wiki/Foreach_loop

Bibliografia e referências

DOWNEY, A. B. Listas. *In*: **Pense em Python**. São Paulo: Editora Novatec, 2016. cap. 10. Disponível em: <https://penseallen.github.io/PensePython2e/10-listas.html>. Acesso em 21 de fev. 2021.

DOWNEY, A. B. Tuplas. *In*: **Pense em Python**. São Paulo: Editora Novatec, 2016. cap. 12. Disponível em: <https://penseallen.github.io/PensePython2e/12-tuplas.html>. Acesso em 21 de fev. 2021.

PROGRAMIZ, L. Python for Loop. 2017. Disponível em: <https://www.programiz.com/python-programming/for-loop>. Acesso em: 14 fev. 2021.

PSF. **Tipos Embutidos: Operações Comuns de Sequências**. 2021a. Disponível em: <https://docs.python.org/3/library/stdtypes.html#common-sequence-operations>. Acesso em: 24 fev. 2021.

PSF. **Tipos Embutidos: Tipos Sequências Mutáveis**. 2021b. Disponível em: <https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types>. Acesso em: 24 fev. 2021.

STURTZ, J. Lists and Tuples in Python. **Real Python**, 2018. Disponível em: <https://realpython.com/python-lists-tuples/>. Acesso em: 21 fev. 2021.

WEBER, B. Defining Main Functions in Python. **Real Python**, 2019. Disponível em: <https://realpython.com/python-main-function/>. Acesso em: 20 fev. 2021.