10

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

10

Operações básicas com sequências

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

Resumo

Nesta aula os objetivos são: (I) explorar o funcionamento do Python em relação aos tipos de dados e sua representação em memória; (II) conhecer as principais funções e métodos para trabalhar com sequências; (III) compreender os tipos de passagem de argumentos para funções e como são aplicados em Python.

10.1. Motivação

Para manipularmos dados em sequências de maneira eficaz, é importante entender quais operações estão disponíveis para todas as sequências e quais são específicas para determinado tipo. Também é necessário aprender o que são objetos e como Python trata os objetos criados na memória, inclusive na passagem de argumentos para funções. Com esse conhecimento, poderemos escolher abordagens mais adequadas para cada problema.

10.2. Introdução

Até o momento, vimos que, em Python, uma possível classificação das sequências é em relação a sua mutabilidade, isto é, alguns tipos de sequências podem ter seus itens modificados e outros não. Portanto, veremos neste capítulo quais operações são comuns a todos os tipos de sequência e quais são válidas apenas para sequências mutáveis.

Também veremos que muitas dessas operações são feitas por meio de métodos. Por isso, será necessário introduzir o conceito de objeto e de método, além da relação entre eles e como isso se aplica em Python.

Por fim, estudaremos o comportamento dos objetos quando passados como argumentos para funções integradas, importadas ou definidas pelo programador.

10.3. Métodos e objetos em Python

No início de nossos estudos, aprendemos que a linguagem Python é multiparadigma, isto é, pode trabalhar simultaneamente com diferentes paradigmas de programação, como *procedural*, *funcional* e *orientado a objetos*. Também foram abordados os diferentes tipos de dados (*int*, *float*, *bool*, *string*, etc.), mas não havíamos abordado que em Python tudo é um objeto, desde números até listas e funções.

Esse conceito é importante para entendermos duas coisas: 1) o que são métodos e; 2) como objetos se comportam quando passados como argumentos para funções.

Podemos pensar em um objeto como uma entidade que agrupa características e comportamentos, e que tem consciência de si mesmo. Os comportamentos são o que chamamos de métodos, e são análogos às funções. Por hora, aprenderemos apenas sobre os objetos que existem por padrão em Python. O que precisamos saber sobre o assunto é:

- Cada objeto possui uma identidade única em Python, como um RG ou CPF;
- Podemos usar os métodos de um objeto para acessar ou alterar suas características, dependendo do tipo do objeto.

É possível obter a identidade de um objeto usando a função integrada `id`, que retorna um número inteiro representando a identidade do objeto. Python garante que esse valor será único enquanto o objeto existir na memória. Na implementação padrão do Python (CPython), esse valor representa o endereço do objeto na memória (PSF, 2021a).

Os métodos em Python podem ser acessados a partir do objeto usando a notação de ponto: `objeto.metodo()`. Talvez você se recorde do método `format`, que pode ser usado para formatar uma *string* de modo semelhante as *f-strings*. Na Codificação 10.1, há um exemplo de uso deste método e da função `id`.

```
>>> texto = 'Olá {}!'
>>> id(texto)
3124235308912
>>> texto
'Olá {}!'
>>> texto.format('mundo')
'Olá mundo!'
```

Codificação 10.1: Exemplo de utilização da função `id` e do método `format`.

Usamos a função integrada `id` para obter a identidade do objeto *string* passado como argumento e, em seguida, o método `format` para acessar o conteúdo desse objeto, através da referência a ele que foi salva na variável `texto`, e retornar uma nova *string* formatada, que foi exibida na *Shell*. Note que há criação de uma nova *string*, pois *strings* são objetos imutáveis, isto é, depois de criadas na memória, não podem ser alteradas.

Confirmamos que a variável `texto` continua referenciando a *string* inicial inspecionando a identidade de seu conteúdo, assim como é possível verificar que o objeto não foi modificado por `format` visualizando o próprio conteúdo, que continua o mesmo.

```
>>> id(texto)
3124235308912
>>> texto
'Olá {}!'
```

Codificação 10.2: Visualização do objeto de *string* inalterado.

Refaça a Codificação 10.1, porém atribuindo o valor de retorno de `format` a outra variável. Aproveite para verificar a identidade do conteúdo da nova variável.

Sabemos que listas são objetos mutáveis, logo suas características podem ser alteradas no mesmo local da memória, sem a necessidade de gerar um novo objeto, diferente da *string* da Codificação 10.1. Por ser mutável, podemos modificar seus itens, ou usar o método `append` para incluir um novo item ao final da lista. Em ambas as situações, podemos verificar que a lista continua o mesmo objeto na memória, apenas com valores diferentes, ou seja, alteramos suas características, mas não sua identidade. Esse comportamento pode ser visto na Codificação 10.3.

```
>>> lista = [1, 2, 3]
>>> lista
[1, 2, 3]
>>> id(lista)
3124235308003
>>> lista[0] = 25
>>> lista
[25, 2, 3]
>>> lista.append(4)
>>> lista
[25, 2, 3, 4]
>>> id(lista)
3124235308003
```

Codificação 10.3: Criação, alterações e inspeções em um objeto `list`.

10.4. Operações comuns de sequências

As operações comuns de sequências (PSF, 2021b) são aquelas que podem ser aplicadas a todos os tipos de sequência do Python, independentemente de serem mutáveis ou não. Portanto, as operações desse grupo se aplicam as listas, tuplas, intervalos e *strings*, e podem ser agrupadas da seguinte forma:

- Pertencimento;
- Concatenação e repetição;
- Indexação e fatiamento;
- Tamanho, item mínimo e item máximo;
- Busca por valor;
- Contagem de ocorrências.

10.4.1. Pertencimento

Verifica se um item pertence à uma sequência. A operação de pertencimento é feita com o operador `in` e a operação inversa pode ser feita com o operador `not in`. Veja na Codificação 10.4.

```
>>> 3 in [1, 2, 5]
False
>>> 5 in range(10)
True
>>> 'z' not in 'banana'
True
```

Codificação 10.4: Utilização do operador de pertencimento.

E para *strings*, há um funcionamento especial deste operador, que pode ser usado também para testes de subsequências. Veja na Codificação 10.5.

```
>>> 'ana' in 'banana'
True
>>> 'ara' not in 'Araraquara'
False
```

Codificação 10.5: Utilização do operador de pertencimento em *strings*.

10.4.2. Concatenação e repetição

A concatenação é feita com o operador `+` e a repetição com o operador `*`. Devido à natureza dos intervalos (*range*), estas operações não são aplicáveis a eles. Veja na Codificação 10.6 e Codificação 10.7 exemplos de uso desses operadores.

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> 'abc' + 'xyz'
'abcxyz'
>>> [30, 40] + [10, 20, 50]
[30, 40, 10, 20, 50]
```

Codificação 10.6: Exemplos de concatenação de sequências.

```
>>> 3 * 'Abc'
'AbcAbcAbc'
>>> 5 * [0]
[0, 0, 0, 0, 0]
>>> 4 * (123,)
(123, 123, 123, 123)
```

Codificação 10.7: Exemplos de repetição de sequências.

A concatenação e a repetição não alteram os objetos, geram um novo. Portanto, não é a forma mais eficiente para adicionar itens a uma sequência. Uma abordagem melhor é usar o método `append` em uma lista, e em seguida, convertê-la para outro tipo de sequência, se necessário.

Tanto a concatenação quanto à repetição passam para a nova sequência as referências aos itens das sequências iniciais, e não uma cópia dos objetos em si. Isso não é um problema quando os itens são imutáveis, mas pode surpreender quem está começando a programar em Python quando os itens são listas. Veja a Codificação 10.8.

```
>>> s = [[2]] * 3
>>> s
[[2], [2], [2]]
```

Codificação 10.8: Repetição de uma lista aninhada.

Aparentemente, criamos uma lista com três sub-listas contendo o número 2. Mas ao alterarmos o primeiro item da primeira sub-lista, as demais sub-listas também são alteradas, como consta na Codificação 10.9.

```
>>> s[0][0] = 5
>>> s
[[5], [5], [5]]
```

Codificação 10.9: Alteração de um item de uma das listas internas.

Teste os exemplos das Codificações 10.9 e 10.10 no Python Tutor para visualizar o que aconteceu e para aprender mais a respeito, leia a discussão na página da PSF em https://docs.python.org/pt-br/3/faq/programming.html#faq-multidimensional-list.

10.4.3. Indexação e fatiamento

A indexação é a forma como acessamos um item de uma sequência, e é feita sempre com o uso de colchetes. Os índices devem obrigatoriamente ser números inteiros e, como já vimos, há dois tipos de índices: os naturais e os negativos. Recordemos por meio da Figura 10.1 como Python define a indexação dos itens de uma sequência e com a Codificação 10.10 como acessar itens específicos de uma sequência.

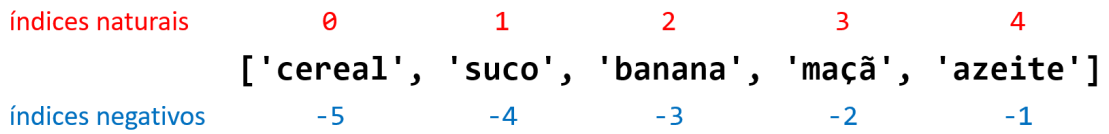

Figura 10.1: Visualização dos índices de uma lista de *strings* em Python.
Fonte: Elaborado pelo autor.

```
>>> texto = 'uma frase qualquer'
>>> texto[4]
'f'
>>> intervalo = range(10, 20)
>>> intervalo[3]
13
>>> tupla = (3.1, 5.8, 7.0)
>>> tupla[0]
3.1
```

Codificação 10.10: Indexação de sequências.

A indexação funciona tanto para acessar um item de uma sequência referenciada por uma variável quanto para acessar o item diretamente da sequência literal, como podemos visualizar na Codificação 10.11.

```
>>> 'abc'[0]
'a'
>>> range(2, 5)[1]
3
>>> [40, 50, 60][2]
60
```

Codificação 10.11: Indexação de sequências literais.

Além de acessar um único item de uma sequência, o Python também permite o que chamamos de fatiamento, quando definimos um intervalo de índices entre colchetes para obter uma "fatia" da sequência, ou seja, uma subsequência.

E, para realizar o fatiamento, devemos passar os índices separados por dois pontos entre os colchetes, seguindo uma notação semelhante à usada em intervalos (*range*): `sequencia[início:fim:passo]`. A Codificação 10.12 contém exemplos de fatiamento com listas, porém se aplicam aos demais tipos de sequências igualmente.

Um fatiamento não altera a sequência fatiada, apenas gera uma nova sequência do mesmo tipo da original e contendo apenas os itens (uma cópia das referências a eles) conforme a definição dada. Observe que, assim como na criação de intervalos, o item correspondente ao índice `fim` não será incluído no fatiamento.

```
>>> lista = [30, 12, 13, 41, 15, 6, 78, 44, 19]
>>> lista[2:4]     # se omitido, passo é 1
[13, 41]
>>> lista[1::3]    # se omitido, fim é len(lista)
[12, 15, 44]
>>> lista[:3]      # se omitido, início é 0
[30, 12, 13]
>>> lista[5:1]     # se fim <= início, a fatia é vazia
[]
>>> lista[5:1:-1]  # se passo é negativo, inverte a ordem da fatia
[6, 15, 41, 13]
>>> lista[:]       # fatia correspondente à lista inteira
[30, 12, 13, 41, 15, 6, 78, 44, 19]
```

Codificação 10.12: Fatiamento de sequências.

10.4.4. Tamanho, soma, mínimo e máximo

Python disponibiliza quatro funções integradas bastante úteis para lidar com sequências, veja na Tabela 10.1 quais são e alguns exemplos na Codificação 10.13.

Tabela 10.1: Funções integradas `len`, `sum`, `min` e `max`.

Função	Descrição
`len(s)`	Retorna o comprimento (tamanho) da sequência `s`.
`sum(s)`	Retorna a soma dos itens da sequência `s`.
`min(s)`	Retorna o item de valor mínimo da sequência `s`.
`max(s)`	Retorna o item de valor máximo da sequência `s`.

```
>>> lista = [30, 12, 13, 41, 15, 6, 78, 44, 19]
>>> len(lista)
9
>>> len(range(3, 10, 2))
4
>>> sum(lista)
258
>>> max(lista)
78
>>> min('Python')   # Por que a menor letra é  'P' e não 'h'?[1]
'P'
```

Codificação 10.13: Utilização das funções integradas `len`, `sum`, `min` e `max`.

[1] Os caracteres são comparados por seus códigos na tabela Unicode e, neste caso, letras latinas maiúsculas antecedem as minúsculas. Para recordar basta usar as funções `ord(caractere)` e `chr(código)`.

10.4.5. Busca por valor

Para buscar o índice de um valor em uma sequência, podemos usar o método `index`, que recebe como argumento o valor buscado e retorna a posição da primeira ocorrência encontrada. Caso não encontre, levanta um erro de valor, indicando que o valor não foi encontrado. Veja exemplos na Codificação 10.14.

```
>>> texto = 'Python Para Pessoas!'
>>> texto.index('P')
0
>>> texto.index('p')
(...)
ValueError: substring not found[2]
```

Codificação 10.14: Utilização do método `index`.

10.4.6. Contagem de ocorrências

Para contar quantas vezes um valor ocorre em uma sequência, usamos o método `count`, que recebe o valor buscado e percorre a sequência inteira, retornando no final quantas vezes aquele valor foi encontrado. Veja exemplos na Codificação 10.15.

```
>>> lista = [1, 2, 3, 41, 5, 7, 3, 41, 41]
>>> lista.count(41)
3
>>> lista.count(200)
0
```

Codificação 10.15: Utilização do método `count`.

10.5. Laços implícitos

Você deve ter reparado que várias funções e métodos aplicados em sequências precisam, para executar corretamente sua funcionalidade, percorrer a sequência. Isso evidencia que há um laço implícito, que são invisíveis por não acessarmos diretamente o código dessas funções e métodos, apenas a usamos. Logo, são operações custosas!

Note que esse custo de percorrer a sequência não ocorre em operações onde um item da sequência é acessado diretamente como, por exemplo, quando um índice é passado para uma função/método indicando a exata e única posição que será acessada.

Para problemas simples, executar algumas operações com laços implícitos não é tão relevante, mas quando trabalhamos com sequências grandes ou quando essas operações são repetidas diversas vezes, esse descuido pode acarretar em um programa que consome recursos desnecessariamente, como tempo de processamento e memória.

[2] O texto na mensagem de erro varia de acordo com o tipo de sequência.

10.6. Operações de sequências mutáveis

Além das operações comuns, nas sequências mutáveis existem mais operações possíveis (PSF, 2021c):

- Substituição de itens da sequência;
- Inclusão de itens na sequência;
- Remoção de itens da sequência;
- Inversão da sequência.

Das sequências que estudamos até agora, as operações desse tópico se aplicam apenas às listas. É importante observar que estes métodos operam por efeito colateral, isto é, eles alteram o objeto da sequência internamente e não retornam valor útil.

10.6.1. Substituição de itens da sequência

A substituição de itens em uma sequência é feita por meio da atribuição direta à uma de suas posições. Veja exemplos na Codificação 10.16.

```
>>> lista = [41, 5, 7]
>>> lista[1] = 200
>>> lista
[41, 200, 7]
```

Codificação 10.16: Alteração de um item em uma sequência mutável.

10.6.2. Inclusão de itens na sequência

A inclusão de itens em uma sequência pode ser feita de três formas, cada uma mais adequada a depender da localização e quantidade de itens que devem ser incluídos.

- `s.insert(i, x)` - insere `x` como item na posição `i` da sequência `s`;
- `s.append(x)` - adiciona `x` como item no final da sequência `s`;
- `s.extend(t)` - adiciona, um a um, os itens da sequência `t` ao final de `s`.

Note na Codificação 10.17 que o inteiro 25 foi incluído na posição 1, deslocando o antigo item desta posição, e os demais itens sucessores, uma posição para a direita.

```
>>> lista = [5, 3, 78]
>>> lista.insert(1, 25)
>>> lista
[5, 25, 3, 78]
```

Codificação 10.17: Inserção se um item em uma posição específica da sequência.

Na Codificação 10.18, o número 100 foi adicionado ao final da lista, assim como a lista [1, 2, 3], que também foi adicionada inteiramente com um único item.

```
>>> lista = [15, 10, 99]
>>> lista.append(100)
>>> lista
[15, 10, 99, 100]
>>> lista.append([1, 2, 3])
[15, 10, 99, 100, [1, 2, 3]]
```

Codificação 10.18: Adição de um item ao final da sequência.

Na Codificação 10.19, vemos que a lista original foi estendida com cada um dos itens da tupla (1, 2, 3). A sequência que recebe os itens precisa ser mutável, portanto uma lista, mas a sequência que cede os itens pode ser de qualquer tipo.

```
>>> lista = [15, 10, 99]
>>> lista.extend((1, 2, 3))
[15, 10, 99, 1, 2, 3]
```

Codificação 10.19: Extensão de uma sequência (mutável) com itens de outra sequência.

10.6.3. Remoção de itens da sequência

Há três formas de um item ser removido de uma sequência:

- `del s[i]` - exclui o item de índice `i` na sequência `s`;
- `s.pop(i)` - exclui o item de índice `i` na sequência `s`, e retorna seu valor;
- `s.remove(x)` - remove o primeiro item de valor `x` da sequência `s`.

Veja na Codificação 10.20 o funcionamento do comando `del`.

```
>>> lista = [15, 10, 99]
>>> del lista[2]
>>> lista
[15, 10]
```

Codificação 10.20: Exclusão de item de uma sequência com o comando `del`.

Observe na Codificação 10.21 que, diferentemente do comando `del`, o método `pop` retorna o item excluído da lista, que é exibido na *Shell* do Python.

```
>>> lista = [15, 10, 99]
>>> lista.pop(0)
15
>>> lista
[10, 99]
```

Codificação 10.21: Exclusão de item de uma sequência com método `pop`.

E na Codificação 10.22, usamos o método `remove` para buscar e remover um item em função do seu valor.

```
>>> lista = [15, 10, 99]
>>> lista.remove(10)
>>> lista
[15, 99]
```

Codificação 10.22: Exclusão de itens de uma sequência com método `remove`.

É importante observar que nas duas primeiras formas, com `del` e `pop`, o item é diretamente acessado por meio de seu índice, portanto é uma operação com custo computacional constante, independente do tamanho da lista. Já no método `remove`, a lista é percorrida até que o valor seja encontrado ou ela termine, portanto há um laço implícito e o custo computacional cresce proporcionalmente ao tamanho da lista.

10.6.4. Inversão da sequência

Para inverter os itens de uma sequência alterando os dados no lugar, isto é, no próprio objeto, usamos o método `reverse`, como mostrado na Codificação 10.23.

```
>>> lista = [10, 20, 30, 40]
>>> lista.reverse()
>>> lista
[40, 30, 20, 10]
```

Codificação 10.23: Inversão de uma sequência com o método `reverse`.

10.7. Operações comuns em *strings*

As *strings*, assim como as listas e tuplas, possuem seu próprio conjunto de métodos. A lista completa pode ser vista na documentação do Python (PSF, 2021d), aqui veremos apenas os dois métodos para converter *strings* em listas e vice-versa.

Para separar uma *string* em uma lista de *strings* usamos o método `split`, que recebe um separador (*string*) e retorna uma lista, como mostrado na Codificação 10.24.

```
>>> '1;2;3;4;5'.split(';')
['1', '2', '3', '4', '5']
```

Codificação 10.24: Utilização do método `split` com um argumento.

Caso o método seja chamado sem argumentos, irá apresentar um comportamento especial, separando a *string* em todos os caracteres que representam espaços em branco (espaço, tabulação e quebra de linha) e eliminando espaços em branco consecutivos.

Observe na Codificação 10.25 que ao passarmos um espaço em branco como argumento, Python coloca uma *string* vazia entre cada espaço adjacente na *string*, mas quando não passamos argumento, ele automaticamente desconsidera os espaços extras.

```
>>> 'frase  com     espaços  '.split(' ')
['frase', '', 'com', '', '', '', '', 'espaços', '', '']
>>> 'frase  com     espaços  '.split()
['frase', 'com', 'espaços']
```

Codificação 10.25: Utilização do método `split` sem argumentos.

O inverso é feito com o método `join`, que recebe como argumento uma sequência de *strings* e gera uma nova *string* com os itens do argumento concatenados e separados pelos caracteres da *string* em que foi chamado, como na Codificação 12.26.

```
>>> '_'.join(['1', '2', '3', '4', '5'])
'1_2_3_4_5'
>>> '_sep_'.join(['a', 'b', 'c'])
'a_sep_b_sep_c'
>>> ''.join(['a', 'b', 'c'])   # separador é uma string vazia
'abc'
>>> ' '.join(['a', 'b', 'c'])  # separador é um espaço
'a b c'
```

Codificação 10.26: Utilização do método `join` para unir uma lista de *strings*.

10.8. Desempacotamento de sequências

O desempacotamento ocorre quando os itens de uma sequência são atribuídos a diversas variáveis em uma única atribuição. Por isso, também é uma operação conhecida como *atribuição paralela*. Há na Codificação 10.27 exemplos de desempacotamento.

```
>>> x, y, z = [15, 10, 99]
>>> print(f'x = {x} | y = {y} | z = {z}')
x = 15 | y = 10 | z = 99
>>> a, b, c = (10, 'Impacta', True)
>>> print(f'a = {a} | b = {b} | c = {c}')
a = 10 | b = Impacta | c = True
```

Codificação 10.27: Desempacotamento de uma sequência em variáveis.

O desempacotamento também funciona com os demais tipos de sequência que vimos, como tuplas, intervalos e *strings*. Será gerado um erro de execução se o número de variáveis à esquerda for diferente do número de itens à direita.

Uma forma simples de realizar a leitura de diversos valores dispostos em uma única linha é com a combinação do método `split` e do desempacotamento.

```
nome, idade, peso = input().split()     # desempacotamento de lista.
idade, peso = int(idade), float(peso)   # desempacotamento de tupla.
```

Codificação 10.28: Leitura de diversos valores dispostos em apenas uma linha.

A variável `nome` não precisou de uma nova atribuição, pois o tipo de seu valor já estava correto no primeiro desempacotamento, afinal seu conteúdo deve ser uma *string*.

Na codificação 10.29, vemos um exemplo desse desempacotamento sendo feito diretamente para os parâmetros de uma função, que é feito com o uso de um asterisco antes do argumento passado à função.

```
def teste(a, b, c):
    print(f'a = {a} | b = {b} | c = {c}')

lista = [1, 'banana', True]
teste(*lista)
```

Codificação 10.29: Desempacotamento para parâmetros de uma função.

O desempacotamento de `lista` na chamada à `teste` equivale à Codificação 10.30, em que cada item da lista é passado individualmente como argumento.

```
teste(lista[0], lista[1], lista[2])
```

Codificação 10.30: Equivalência ao desempacotamento da Codificação 10.29.

Note que usar desempacotamento para passar argumentos a uma função não tem relação com a forma como ela foi definida, pois a função receberá os argumentos já desempacotados, ou seja, uma chamada comum. Porém, é necessário garantir que o desempacotamento resulte em quantidade de argumentos igual a de parâmetros.

Entretanto, há funções com quantidade variável de argumentos, como a função `print`. Não entraremos em detalhes de como construir essas funções, mas você poderá aprender mais sobre elas em Mastromatteo (2019).

VOCÊ SABIA?

É possível unir os dois recursos de Python, uma função com quantidade variável de argumentos e o desempacotamento de uma sequência para passar argumentos à função. Assim, podemos, por exemplo, exibir todos os itens de uma lista só com uma instrução:

```
>>> lista = [2, 3, 5, 7]
>>> print(*lista)
```

10.9. Tipos de passagem de argumentos

Ao chamar uma função é comum que existam duas alternativas para como os argumentos serão passados: (1) passando o valor diretamente, desta forma o parâmetro da função receberá uma cópia do valor original ou (2) passando uma referência para o local da memória onde o valor está, assim o parâmetro da função receberá o endereço do valor original. Na Figura 10.2 há uma representação simplificada de como um programa “enxerga” as variáveis criadas na memória com base na Codificação 10.31.

```
n = 7
valido = True
letra = 'r'
```

Codificação 10.31: Criação de variáveis em um programa em Python.

Programa

Identificadores	Endereço na memória
n	0x45c1f3
valido	0x0034cc
letra	0x14faf1

Memória do computador

Endereço na memória	Conteúdo no endereço
0x45c1f3	7
0x0034cc	True
0x14faf1	'r'

Figura 10.2: Representação simplificada dos dados na memória do computador. Fonte: Elaborado pelo autor.

Há linguagens em que é possível escolher a forma como os argumentos serão passados à função, por valor ou por referência. Veja a Codificação 10.32, que define uma função com um parâmetro `x`, e em seguida faz uma chamada a essa função colocando como argumento a variável `n`, criada na Codificação 10.31. O bloco de código da função é irrelevante para nossa análise, por isso foi suprimido.

```
def my_function(x):
    # bloco de código da função
my_function(n)
```

Codificação 10.32: Chamada à uma função com passagem de argumento por valor.

Se a passagem do argumento na chamada da Codificação 10.32 fosse por valor, o parâmetro `x` da função receberia uma cópia do valor que está no argumento `n`. A cópia seria armazenada em outro local na memória e, consequentemente, teria outro endereço. Lembre-se que `x` está em uma "tabela" local da função. Veja a Figura 10.3.

Escopo local: `my_function`

Identificadores	Endereço na memória
x	0x230f7a

Memória do computador

Endereço na memória	Conteúdo no endereço
0x230f7a	7

Figura 10.3: Representação simplificada do escopo local da função e da memória do computador na passagem por valor. Fonte: Elaborado pelo autor.

Observe que o conteúdo associado à `x` seria idêntico ao conteúdo associado à `n`, mas os endereços dos conteúdos seriam diferentes. Portanto, ao alterarmos o valor associado à variável local `x`, isso não afetaria o conteúdo associado à variável global `n`.

Agora, se a passagem desse argumento for por referência, o parâmetro `x` receberá uma referência ao conteúdo original, e não será feita uma cópia do conteúdo original na memória. Veja a representação na Figura 10.4.

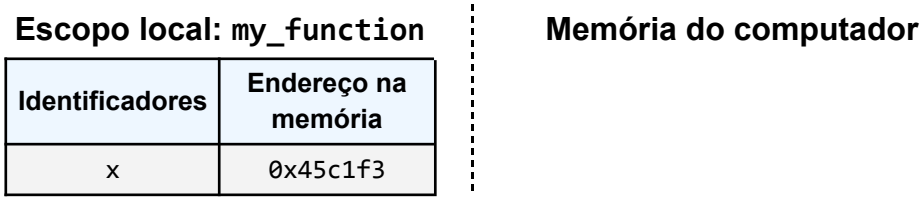

Figura 10.4: Representação simplificada do escopo local da função e da memória do computador na passagem por referência. Fonte: Elaborado pelo autor.

Esse tipo de passagem tem a vantagem de economizar recursos, tanto memória quanto processamento, por evitar a criação de cópias de objetos. Porém, ao alterarmos o conteúdo associado à variável `x`, implicitamente alteramos o valor associado à variável global `n`. A isso damos o nome de *efeito colateral*, pois ao executar a função, valores externos a ela serão modificados. Comportamento útil quando bem aplicado, mas que pode gerar erros difíceis de rastrear se ser mal utilizado.

10.9.1. Passagem de argumentos em Python

Por uma decisão de projeto, Python não permite a escolha de como argumentos são passados às funções. Visando a economia de recursos, dentre outras possíveis razões, em Python todos os argumentos são passados por referência, mas isso não significa que sempre teremos o comportamento explicado anteriormente[3]. Há um misto de passagem por valor e por referência que dependerá da mutabilidade do argumento.

Quando um objeto é imutável, Python não permite que o conteúdo do objeto naquele endereço de memória seja alterado, então ao modificarmos o valor da variável, ao invés de alterar o conteúdo do "espaço" na memória, Python criará uma nova entrada na memória e associará essa nova referência à variável. Portanto, objetos imutáveis têm um comportamento análogo à passagem de argumentos por valor.

Já em objetos mutáveis, o conteúdo do objeto pode ser alterado a partir de qualquer variável que o referencie, inclusive os parâmetros da função. Caso o objeto alterado precise de mais memória, por exemplo uma lista à qual adicionamos novos itens, o Python gerenciará essa alocação de memória conforme necessário, sem trocar a associação entre o identificador e a referência para o objeto alterado[4]. Sendo assim,

[3] Se esse fosse o caso, é possível que os programas em Python teriam muitos *bugs*, e a linguagem dificilmente teria ganho a notoriedade que possui hoje em tantas áreas diferentes da programação.
[4] Isso só ocorre quando usamos funções ou métodos especiais que acessam os itens do objeto. Se fizermos uma atribuição direta de outro valor à variável, a referência será sobrescrita normalmente, independente da mutabilidade do objeto.

objetos mutáveis se comportam de modo análogo à passagem de argumentos por referência[5]. A Figura 10.5 exibe a mutabilidade dos principais tipos em Python.

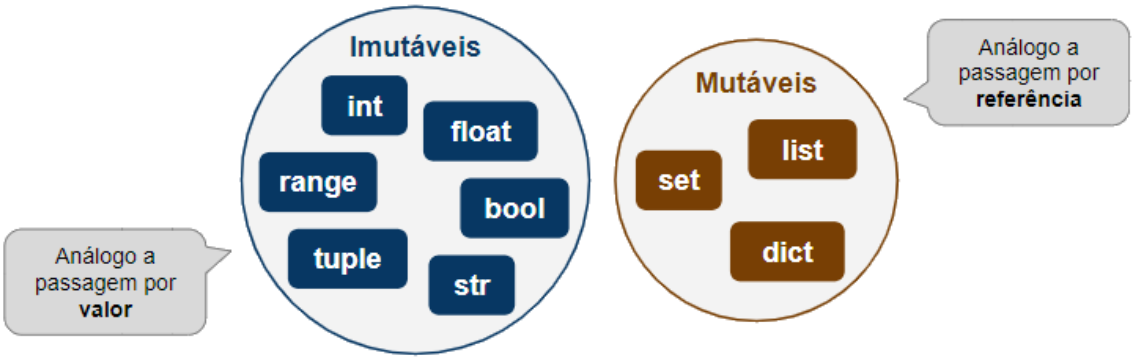

Figura 10.5: Mutabilidade dos principais tipos em Python. Fonte: Elaborado pelo autor.

A passagem de argumentos adotada pelo Python é também conhecida como "passagem por objeto". Lembre-se que vimos no início do capítulo que tudo em Python é um objeto. Então, ao chamarmos uma função, Python passará as referências dos objetos dispostos como argumentos para os parâmetros da função, sem criar cópias.

Veja na Codificação 10.33 uma função que recebe um inteiro `n` e uma sequência mutável `s` como argumentos e realiza operações nesses objetos. Execute o código no Python Tutor para observar o que acontece com os objetos na memória a cada instrução. Para deixar mais claro o tratamento dos números inteiros como objetos, antes de executar o código altere a opção de renderização, conforme a Figura 10.6.

```
def teste(n, s):
    n = n + 2
    s[0] = s[0] + 1

n1 = 1
lista = [1]

teste(n1, lista)

print('fim')
```

Codificação 10.33: Código para exemplificação do comportamento do Python em relação a passagem de argumentos.

Veja na Figura 10.7 a comparação entre o estado da memória antes, durante e após a execução da função `teste`. O valor da variável `n1` não foi alterado, mas o item da `lista` foi alterado de `1` para `2` e essa alteração persiste após a execução da função.

[5] Note que isso não significa que sejamos obrigados a alterar o próprio objeto, é perfeitamente possível criar funções que não possuam efeitos colaterais mesmo quando usadas com objetos mutáveis, como mostrado na Codificação 10.34.

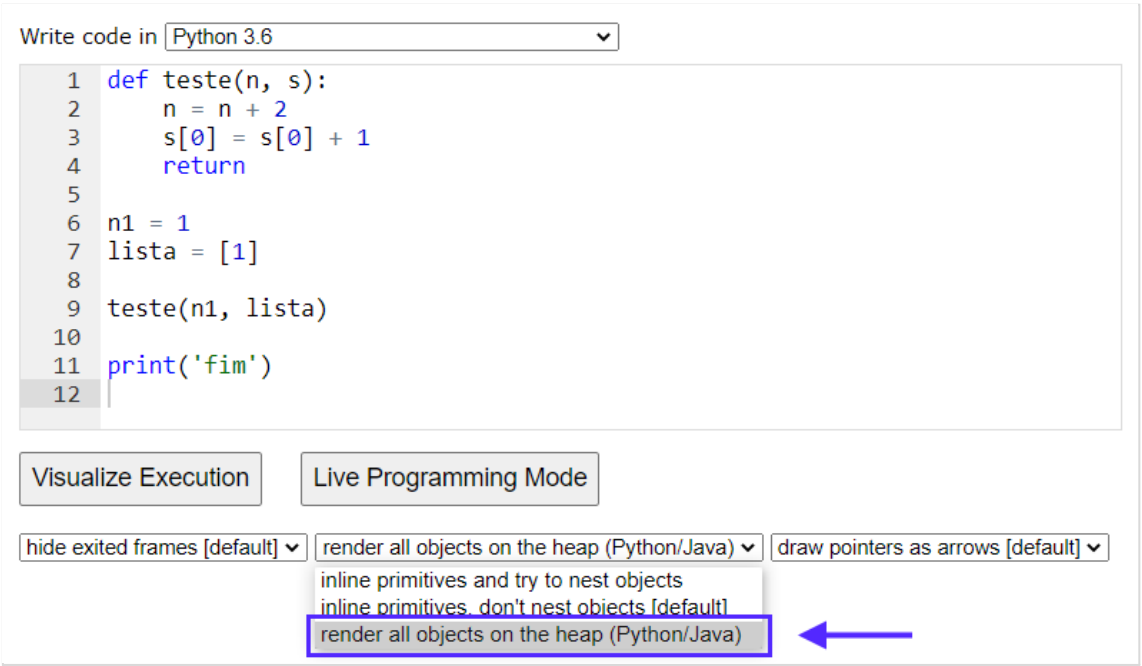

Figura 10.6: Alteração da renderização dos objetos no Python Tutor. Fonte: Elaborado pelo autor.

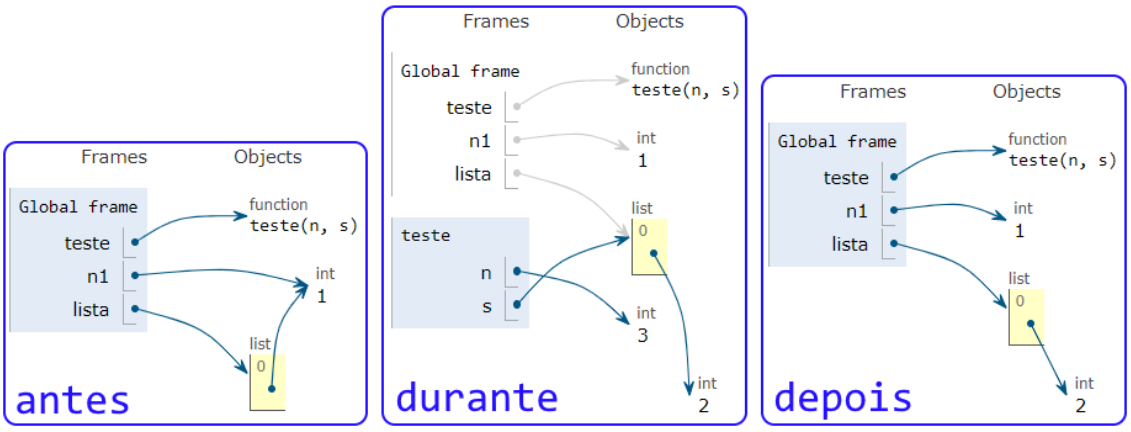

Figura 10.7: Progressão dos estados de memória no decorrer da execução da Codificação 10.33 no Python Tutor. Fonte: Elaborado pelo autor.

Execute o código da Codificação 10.34 no Python Tutor. Se ainda estiver na mesma janela em que testou a Codificação 10.33, ao editar o código e antes de visualizar a execução, retorne a renderização dos objetos na memória para a configuração padrão: *inline primitives, don't nest objects [default]*. Mesmo que omita alguns detalhes, essa configuração torna mais simples a visualização.

Observe na Codificação 10.34 que na função com efeitos colaterais não há retorno de valor, pois as alterações são feitas no objeto original, enquanto na função sem efeitos colaterais, atribuímos o valor retornado a uma variável para guardá-lo e usá-lo.

```
# Função com efeitos colaterais
def converte_1(lista):
    for i in range(len(lista)):
        lista[i] = int(lista[i])

# Função sem efeitos colaterais
def converte_2(lista):
    nova_lista = []
    for item in lista:
        nova_lista.append(int(item))
    return nova_lista

numeros_1 = ['10', '20', '30', '40']
numeros_2 = ['10', '20', '30', '40']

converte_1(numeros_1)
numeros_3 = converte_2(numeros_2)

print(f'lista 1: {numeros_1}', f'lista 2: {numeros_2}',
      f'lista 3: {numeros_3}', sep='\n')
```

Codificação 10.34: Programa para visualização de funções com e sem efeitos colaterais.

Bibliografia e referências

Docstring Guide. **Numpydoc** 2021. Disponível em: <https://numpydoc.readthedocs.io/en/latest/format.html>. Acesso em: 25 jan. 2021.

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

MASTROMATTEO, D. Python args and kwargs: Demystified. **Real Python**, 2019. Disponível em: <https://realpython.com/python-kwargs-and-args/>. Acesso em: 29 mar. 2021.

PSF. **Built-in Functions: id**. 2021a. Disponível em: <https://docs.python.org/pt-br/3/library/functions.html#id>. Acesso em: 13 mar. 2021.

PSF. **Built-in Types: Common Sequence Operations**. 2021b. Disponível em: <https://docs.python.org/3/library/stdtypes.html#common-sequence-operations>. Acesso em: 03 fev. 2021.

PSF. **Built-in Types: Mutable Sequence Types**. 2021c. Disponível em: <https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types>. Acesso em: 03 fev. 2021.

PSF. **Built-in Types: String Methods**. 2021d. Disponível em: <https://docs.python.org/3/library/stdtypes.html#string-methods>. Acesso em: 03 fev. 2021.