Memória e passagem de parâmetros

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Até o momento, quais eram as nossas preocupações no projeto de programas?

- Identificar o problema
- Resolver o problema (com código bem escrito e testado)

Agora vamos discutir outro aspecto importante no projeto de programas: o uso de recursos, especificamente o uso de memória.

Memória

A memória é um recurso compartilhado entre os diversos programas que estão em execução (processos) em um computador. O sistema operacional faz a gerência de memória entre os diversos processos e garante que cada processo só tenha acesso a memória destinada a ele.

Cada processo também precisa gerenciar a sua própria memória.

Algumas linguagens como Python, Java e Go, fazem a gerência automática da memória. Outras linguagens, como C, requerem que o programador faça a gerência da memória de forma explícita (manual).

Cada estratégia de gerência de memória tem vantagens e desvantagem, mas o ponto principal é a facilidade de programação versus o controle. Vocês vão aprender mais sobre isso ao longo do curso!

A gerência de memória requer basicamente duas operações: a alocação e a desalocação de memória.

O que significa alocar memória? É reservar um espaço de memória para ser usado de uma determinada forma.

O que significa desalocar memória? É devolver para o sistema um espaço de memória que havia sido alocado previamente para que ele possa ser usada de outra forma.

Nos programas que fizemos, em que momento o Python aloca memória?

- Quando um valor é criado
- Quando um elemento é adicionado a uma lista e a memória reservada para a lista não é suficiente

Nos programas que fizemos, em que momento o Python desaloca memória?

- Quando um valor não é mais necessário

Variáveis

Vimos anteriormente que uma variável em Python é uma referência para uma célula de memória que armazena um valor.

Agora vamos explorar esse fato com mais detalhes e observar alguns resultados que podem ser surpreendentes.

Qual é o valor de x e y após a execução do seguinte trecho de código?

```
>>> x = 10
>>> y = x
>>> y = y + 3

>>> x
10
>>> y
13
```

Qual é o valor de x e y após a execução do seguinte trecho de código?

```
>>> x = [5, 7]
>>> y = x
>>> y[1] = 3

>>> x
[5, 3]
>>> y
[5, 3]
```

No exemplo da esquerda, após a execução de `y = x`, `x` e `y` referenciam a mesma célula de memória (que armazena o valor `10`). A operação `y + 3` **cria um novo** valor que é armazenado em uma nova célula, que passa a ser referencia por `y` após `y = y + 3`.

Veja esse processo no Python Tutor.

No exemplo da direita, após a execução de `y = x`, `x` e `y` referenciam a mesma célula de memória (que armazena o valor `[5, 1]`). A operação `y[1] = 3` **altera** o valor armazenado na célula de memória para `[5, 3]`.

Veja esse processo no Python Tutor.

Quando uma célula de memória pode ser acessada usando mais do que uma variável (nome), dizemos que existem **apelidos** para a célula de memória.

Parâmetro e apelidos

Quando uma variável é passada como parâmetro para uma função, um apelido é criado.

```
>>> def soma1(x: int):
...     x = x + 1
>>> a = 20
>>> soma1(a)
>>> a

20
```

Quando **soma1** inicia a execução, **a** e **x** referenciam a mesma célula de memória. A instrução `x = x + 1` gera um **novo valor** (`21`) que é armazenado em uma **nova célula** de memória e **x** passa a referenciar essa nova célula. **a** continua referenciado a mesma célula de memória.

```
>>> def concatena1(x: list[int]):
...     x.append(1)
>>> a = [5, 4]
>>> concatena1(a)
>>> a

[5, 4, 1]
```

Quando **concatena1** inicia a execução, **a** e **x** referenciam a mesma célula de memória. A instrução `x.append(1)` **altera a célula** de memória referenciada por **x** adicionando o valor **1**. **a** continua referenciado a mesma célula de memória (que foi alterada).

Os apelidos podem deixar o código mais difícil de ler, mas em algumas situações eles são necessários.

Suponha que queremos projetar uma função que inverta a ordem dos elementos de uma lista, isto é, coloque o último em primeiro, o penúltimo em segundo, e assim por diante.

Como podemos proceder?

Temos duas opções:

1) Fazer uma função que crie uma nova lista com os elementos em ordem invertida
2) Modificar a própria lista alterando a ordem dos elementos

Em geral, criar uma nova lista é mais fácil, mas acarreta no uso extra de memória. Esta pode ser a única opção se tanto a lista inicial quando a lista invertida são utilizadas posteriormente.

Se a lista na ordem inicial não é necessária após a chamada da função, então podemos modificar a própria lista, o que pode ser mais complicado, mas evita o uso de memória extra.

Exemplo: inverte

Vamos primeiro projetar a função que cria uma nova lista.

Exemplo: inverte

```
def inverte(lst: list[int]) -> list[int]:
    '''
    Cria uma nova lista com os elementos de
    *lst* em ordem inversa, isto é, o último
    aparece como primeiro, o penúltimo com
    segundo, e assim por diante.

    Exemplos
    >>> inverte([])
    []
    >>> inverte([8, 6, 1, 4])
    [4, 1, 6, 8]
    '''
```

Qual é a ideia para implementar a função? Percorrer os elementos de `lst` a partir do último e adicionar em uma nova lista.

Vamos escrever o código para uma lista de tamanho fixo e depois generalizar.

```
    # Solução para uma lista de tamanho 4
    r = []
    r.append(lst[3])
    r.append(lst[2])
    r.append(lst[1])
    r.append(lst[0])
    return r
```

Transformando em repetição lógica:

```
    r = []
    for i in range(4):
        r.append(lst[4 - i - 1])
    return r
```

Generalizando para qualquer tamanho:

```
    r = []
    for i in range(len(lst)):
        r.append(lst[len(lst) - i - 1])
    return r
```

Exemplo: inverte

Agora vamos projetar uma função que altera uma lista invertendo a ordem dos elementos.

Exemplo: inverte

```
def invertem(lst: list[int]):
    '''
    Inverte a ordem dos elementos de *lst*,
    isto é, colocando o último elemento na
    primeira posição, o penúltimo na segunda
    posição, e assim por diante.

    Exemplos
    >>> x = []
    >>> invertem(x)
    >>> x
    []
    >>> x = [8, 6, 1, 4, 5]
    >>> invertem(x)
    >>> x
    [5, 4, 1, 6, 8]
    '''
```

De que forma a especificação dessa função é diferente das demais?

Não tem tipo de saída. Por que? A função não vai produzir uma saída e sim o efeito colateral de modificar a lista.

O propósito enfatiza que a lista é modificada.

Os exemplos são especificados em três partes: inicialização dos parâmetros, chamada da função e verificação do efeito.

Exemplo: inverte

```
def invertem(lst: list[int]):
    '''
    Inverte a ordem dos elementos de *lst*,
    isto é, colocando o último elemento na
    primeira posição, o penúltimo na segunda
    posição, e assim por diante.

    Exemplos
    >>> x = [8, 6, 1, 4, 5]
    >>> invertem(x)
    >>> x
    [5, 4, 1, 6, 8]
    '''
```

Qual é a ideia para implementar a função? Trocar o primeiro com o último, o segundo com o penúltimo e assim por diante.

Vamos escrever o código para uma lista de tamanho fixo e depois generalizar.

```
# Vamos escrever a solução para
# uma lista de tamanho 5

# troca lst[0] <-> lst[4]
# lst = [8, 6, 1, 4, 5] -> [5, 6, 1, 4, 8]

t = lst[0]
lst[0] = lst[4]
lst[4] = t

# troca lst[1] <-> lst[3]
# [5, 6, 1, 4, 8] -> [5, 4, 1, 6, 8]

t = lst[1]
lst[1] = lst[3]
lst[3] = t
```

Transformando em repetição lógica:

```
for i in range(2):
    # troca lst[i] <-> lst[5 - i - 1]
```

Exemplo: inverte

```
def invertem(lst: list[int]):
    '''
    Inverte a ordem dos elementos de *lst*,
    isto é, colocando o último elemento na
    primeira posição, o penúltimo na segunda
    posição, e assim por diante.

    Exemplos
    >>> x = [8, 6, 1, 4, 5]
    >>> invertem(x)
    >>> x
    [5, 4, 1, 6, 8]
    '''
```

Qual é a ideia para implementar a função? Trocar o primeiro com o último, o segundo com o penúltimo e assim por diante.

Vamos escrever o código para uma lista de tamanho fixo e depois generalizar.

Transformando em repetição lógica:

```
    for i in range(2):
        # troca lst[i] <-> lst[5 - i - 1]

        t = lst[i]
        lst[i] = lst[5 - i - 1]
        lst[5 - i - 1] = t
```

Generalizando para qualquer tamanho:

```
    for i in range(len(lst) // 2):
        # troca lst[i] <-> lst[len(lst) - i - 1]
        t = lst[i]
        lst[i] = lst[len(lst) - i - 1]
        lst[len(lst) - i - 1] = t
```

Exemplo: insere ordenado

Dado uma lista de números em ordem não decrescente e um valor **v**, projete uma função que modifique a lista inserindo o valor **v** de maneira que o arranjo continue em ordem.

Exemplo: insere ordenado

```
def insere_ordenado(lst: list[int], v: int):
    '''
    Insere *v* em *lst* de maneira que *lst*
    permaneça em ordem não decrescente. Requer
    que *lst* esteja em ordem não decrescente.
    Exemplos
    >>> lst = []
    >>> insere_ordenado(lst, 7)
    >>> lst
    [7]
    >>> insere_ordenado(lst, 3)
    >>> lst
    [3, 7]
    >>> insere_ordenado(lst, 5)
    >>> lst
    [3, 5, 7]
    >>> insere_ordenado(lst, 4)
    >>> lst
    [3, 4, 5, 7]
    '''
```

Qual é a ideia para implementar a função? Colocar `v` no final de `lst` e ir trocando ele de lugar com o antecessor até chegar no “lugar certo”.

Vamos escrever o código para uma lista de tamanho fixo e depois generalizar.

Exemplo: insere ordenado

```
def insere_ordenado(lst: list[int], v: int):
    '''
    Insere *v* em *lst* de maneira que *lst*
    permaneça em ordem não decrescente. Requer
    que *lst* esteja em ordem não decrescente.
    Exemplos
    >>> lst = []
    >>> insere_ordenado(lst, 7)
    >>> lst
    [7]
    >>> insere_ordenado(lst, 3)
    >>> lst
    [3, 7]
    >>> insere_ordenado(lst, 5)
    >>> lst
    [3, 5, 7]
    >>> insere_ordenado(lst, 4)
    >>> lst
    [3, 4, 5, 7]
    '''
    # v = 4
    # lst = [3, 5, 7] -> [3, 5, 7, 4]

    lst.append(v)

    # lst[2] > lst[3], então troca lst[3] <-> lst[2]
    # [3, 5, 7, 4] -> [3, 5, 4, 7]

    t = lst[3]
    lst[3] = lst[2]
    lst[2] = t

    # lst[1] > lst[2], então troca lst[2] <-> lst[1]
    # [3, 5, 4, 7] -> [3, 4, 5, 7]

    t = lst[2]
    lst[2] = lst[1]
    lst[1] = t

    # lst[0] < lst[1], ou seja,
    # v está na posição "certa", então para
    # [3, 4, 5, 7]
```

Exemplo: insere ordenado

```
def insere_ordenado(lst: list[int], v: int):
    '''
    Insere *v* em *lst* de maneira que *lst*
    permaneça em ordem não decrescente. Requer
    que *lst* esteja em ordem não decrescente.
    Exemplos
    >>> lst = []
    >>> insere_ordenado(lst, 7)
    >>> lst
    [7]
    >>> insere_ordenado(lst, 3)
    >>> lst
    [3, 7]
    >>> insere_ordenado(lst, 5)
    >>> lst
    [3, 5, 7]
    >>> insere_ordenado(lst, 4)
    >>> lst
    [3, 4, 5, 7]
    '''
```

Transformando em repetição lógica:

```
lst.append(v)
i = 3
while ...:
    # troca lst[i] <-> lst[i - 1]
    ...
    i = i - 1
```

Completando

```
lst.append(v)
i = 3
while i > 0 and lst[i - 1] > lst[i]:
    # troca lst[i] <-> lst[i - 1]
    t = lst[i]
    lst[i] = lst[i - 1]
    lst[i - 1] = t
    i = i - 1
```

Exemplo: insere ordenado

```
def insere_ordenado(lst: list[int], v: int):
    '''
    Insere *v* em *lst* de maneira que *lst*
    permaneça em ordem não decrescente. Requer
    que *lst* esteja em ordem não decrescente.
    Exemplos
    >>> lst = []
    >>> insere_ordenado(lst, 7)
    >>> lst
    [7]
    >>> insere_ordenado(lst, 3)
    >>> lst
    [3, 7]
    >>> insere_ordenado(lst, 5)
    >>> lst
    [3, 5, 7]
    >>> insere_ordenado(lst, 4)
    >>> lst
    [3, 4, 5, 7]
    '''
```

Generalizando para qualquer tamanho:

```
def insere_ordenado(lst: list[int], v: int):
    lst.append(v)
    i = len(lst) - 1
    while i > 0 and lst[i - 1] > lst[i]:
        # troca lst[i] <-> lst[i - 1]
        t = lst[i]
        lst[i] = lst[i - 1]
        lst[i - 1] = t
        i = i - 1
```

Em Python as variáveis são referências para células de memória que armazenam valores.

Apelidos são variáveis que referenciam a mesma célula de memória.

Quando atribuímos uma variável para outra e quando passamos uma variável como parâmetro para uma função, estamos criando um apelido.

Usamos apelidos (passagem de parâmetro por referência) no projeto de funções que alteram os argumentos (efeito colateral).

Escrevemos o propósito das funções que alteram os argumentos destacando que os argumentos são alterados.

Os exemplos são especificados em três partes: inicialização dos parâmetros, chamada da função e verificação da modificação (efeito colateral).

Na implementação começamos com uma ideia, depois iniciamos a implementação com repetição física de código que concretiza a ideia para uma lista de tamanho fixo, depois transformamos a repetição física em repetição lógica e por fim generalizamos a implementação para listas de qualquer tamanho.