

Memória e passagem de parâmetros

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhável 4.0 Internacional.

<http://github.com/malbarbo/na-programacao>

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.

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 usada de uma determinada forma.

O que significa desalocar memória? É devolver para o sistema um espaço de memória que havia sido alocada previamente para que ela 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

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 alguns resultados que podem parecer 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 referenciada por y após $y = y + 3$.

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]**.

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

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.

Como projetar funções que modificam os seu argumentos?

Generalizando exemplos concretos.

O exemplo a seguir foi desenvolvido em sala.

Exemplo: inverte

```
def inverte(lst: list[int]) -> list[int]:  
    '''  
    Cria uma nova lista invertendo os elementos  
    de *lst*, isto é, o último aparece como  
    primeiro, o penúltimo com segundo, e assim  
    por diante.  
    Exemplos  
    >>> inverte([8, 6, 1, 4, 5])  
    [5, 4, 1, 5, 8]  
    '''  
    r = []  
    for i in range(len(lst)):  
        r.append(lst[len(lst) - i - 1])  
    return r
```

```
def invertem(lst: list[int]):  
    '''  
    Modifica *lst* invertendo a ordem dos  
    elementos, isto é, colocando o último  
    elemento na segunda posição, ...  
    Exemplos  
    >>> x = [8, 6, 1, 4, 5]  
    >>> invertem(x)  
    >>> x  
    [5, 4, 1, 5, 8]  
    '''  
    i = 0  
    j = len(lst) - 1  
    while i < j:  
        t = lst[i]  
        lst[i] = lst[j]  
        lst[j] = t  
        i = i + 1  
        j = j - 1
```

Dado uma lista ordenado 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 feito em sala.

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, 5)  
>>> lst  
[5]  
>>> insere_ordenado(lst, 3)  
>>> lst  
[3, 5]  
>>> insere_ordenado(lst, 4)  
>>> lst  
[3, 4, 5]  
>>> insere_ordenado(lst, 1)  
>>> lst  
[1, 3, 4, 5]  
    '''
```

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