

# Passagem de parâmetros e Recursividade

Programação 1

[pcardoso@ufpa.br](mailto:pcardoso@ufpa.br)

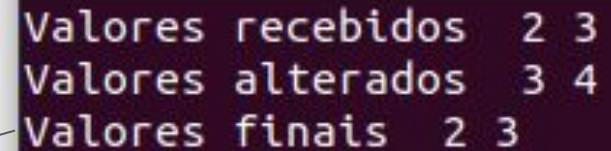
# Um pouco mais sobre modularização

# Subprograma

```
def nomeDoSubprograma (lista de parâmetros):  
    corpo_do_Subprograma
```

# Porque isso acontece?

```
1 def altera (a,b):  
2     print ("Valores recebidos ", a, b)  
3     a+=1  
4     b+=1  
5     print ("Valores alterados ", a, b)  
6  
7  
8 #bloco principal  
9 a,b=2,3  
10 altera(a,b)  
11 print ("Valores finais ", a, b)  
12
```



```
Valores recebidos 2 3  
Valores alterados 3 4  
Valores finais 2 3
```

Dizemos que *a* e *b* da linha 9 são argumentos.


As variáveis *a* e *b* da linha 1 são parâmetros da função.

# Escopo de variáveis

Em python, variáveis em escopo diferentes podem possuir identificadores idênticos.

É importante observar que, embora seja usado o mesmo nome para estas variáveis, elas se referem a endereços de memória completamente diferentes e, portanto, podem armazenar informações completamente diferentes (inclusive tipos de dados diferentes)

- em python, use `id (variavel)` para retornar o identificador de uma variável
- em c++, use `&variavel` para verificar o endereço de uma variável na memória



Fica a dica!

# Escopo de variáveis

No exemplo dado, *a* e *b* definidas no "bloco principal" são diferentes de *a* e *b* da linha 1.

Os parâmetros *a* e *b* da linha 1 recebem uma cópia das variáveis *a* e *b* da linha 9.

Por isso as mudanças feitas na cópia não são vistas no bloco principal.

```
1 def altera (a,b):
2     print ("Valores recebidos ", a, b)
3     a+=1
4     b+=1
5     print ("Valores alterados ", a, b)
6
7
8 #bloco principal
9 a,b=2,3
10 altera(a,b)
11 print ("Valores finais ", a, b)
12
```

# Passagem de parâmetros

Existem dois tipos principais de passagem de parâmetros em funções: passagem **por valor** e passagem **por referência**.

## **Passagem por valor:**

- Em Python, os argumentos são passados por valor. Isso significa que, quando você chama uma função e passa um argumento, uma cópia do valor desse argumento é passada para a função. Qualquer modificação feita dentro da função não afeta o valor original fora da função.
- Exemplo: slide anterior.

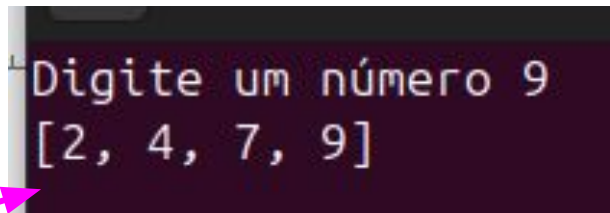
# Passagem de parâmetros

## Passagem por Referência

- Neste caso, as modificações feitas dentro da função afetam o valor original fora da função. Embora Python não suporte a passagem por referência direta, podemos simular esse comportamento passando objetos mutáveis, como listas ou dicionários.

- Exemplo:

```
1. def alteraLista (lista):  
2.     n=int(input("Digite um número "))  
3.     lista.append(n)  
4.  
5. #bloco principal  
6. minhaLista = [2, 4, 7]  
7. alteraLista(minhaLista)  
8. print(minhaLista)
```



```
1 Digite um número 9  
[2, 4, 7, 9]
```



# Resultados múltiplos

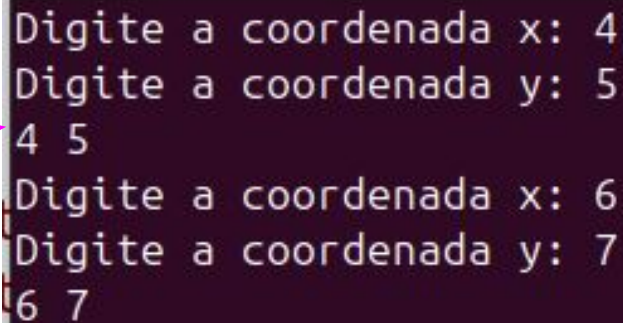
É comum precisarmos devolver mais de um resultado como retorno de um subprograma.

Nestes casos, especificamente no caso da linguagem python, seria necessário retornar uma estrutura de dados ou escrever uma função para cada retorno desejado.

Exemplo:

# Resultados múltiplos

```
1. def lerCoordenadas():
2.     x=int(input("Digite a coordenada x: "))
3.     y=int(input("Digite a coordenada y: "))
4.     return x,y
5.
6. #bloco principal
7. coord = lerCoordenadas()
8. print(coord[0], coord[1])
9.
10. a,b=lerCoordenadas()
11. print(a,b)
```



```
Digite a coordenada x: 4
Digite a coordenada y: 5
4 5
Digite a coordenada x: 6
Digite a coordenada y: 7
6 7
```

- Python usa empacotamento de dados (**tupla**) na linha 7 para receber os 2 valores da função. *Não iremos aprofundar nesse tópico.*
- Outra possibilidade é determinar duas variáveis para receber os valores (linha 10)

# Revisão

Qual o tipo de passagem de parâmetro está sendo aplicada? por valor ou por referência?

Considere  $n=5$ , qual será o valor de saída?

```
1 def fatorial (n):  
2     fat=1  
3     for i in range (1,n+1):  
4         fat*=i  
5     return fat  
6  
7 #bloco principal  
8 n=int(input())  
9 print(fatorial(n))  
10
```

# Relação de recorrência

Uma relação de recorrência é uma equação que define uma função **recursivamente**, usualmente por meio de um ou mais casos bases e um ou mais casos gerais. Por exemplo, a relação de recorrência que define o fatorial de um número inteiro positivo **N**.

$$N! = \begin{cases} 1, & \text{se } N = 0 \\ N * (N-1)!, & \text{se } N > 0 \end{cases}$$

# Recursão e funções

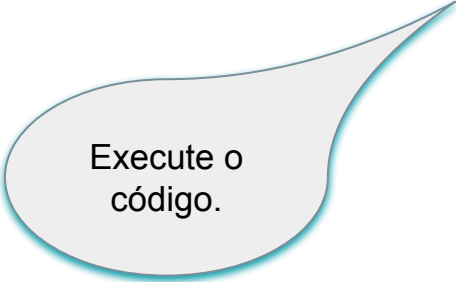
Uma das maneiras de traduzirmos a relação de recorrência:

$$N! = \begin{cases} 1, & \text{se } N = 0 \\ N * (N-1)!, & \text{se } N > 0 \end{cases}$$

em um programa seria por meio da implementação de uma função que chama a si mesma, como no exemplo a seguir.

# Exemplo de função recursiva

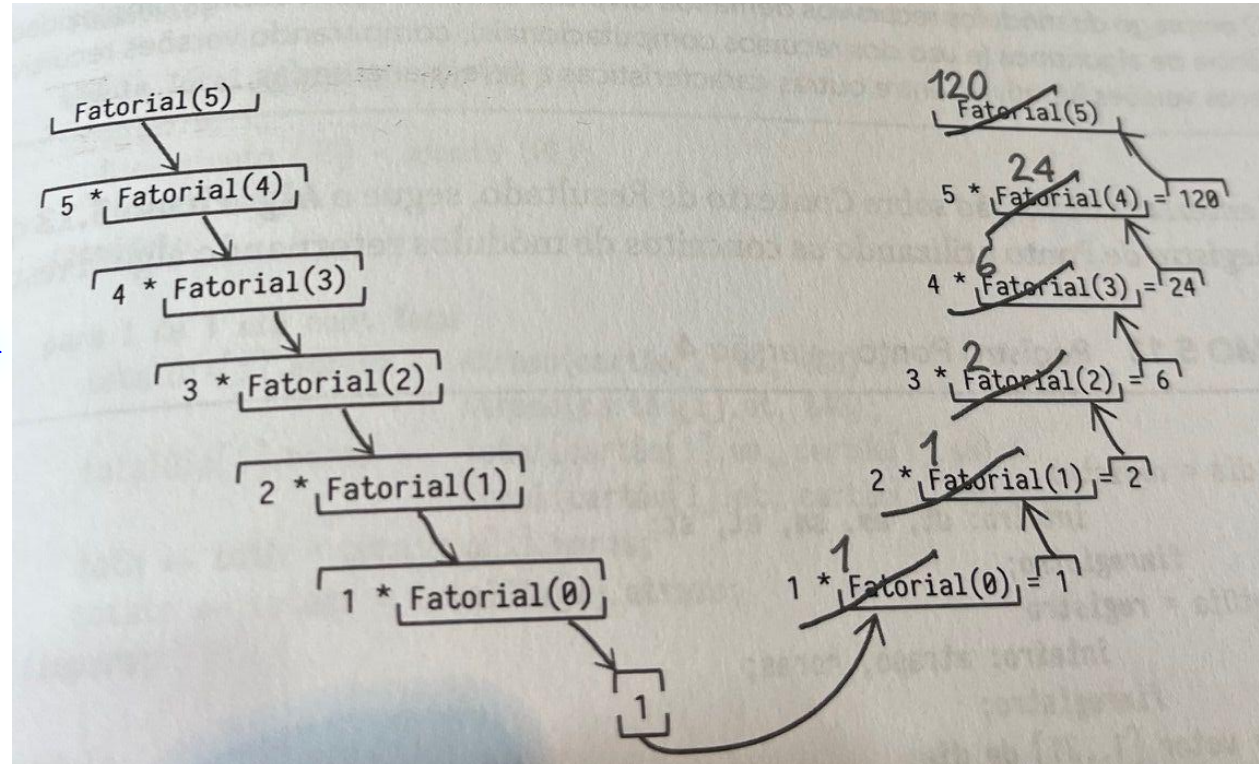
```
1. def fatorialRecursivo (n):  
2.     if n==0:  
3.         return 1  
4.     else:  
5.         return n * fatorialRecursivo (n-1)
```



Execute o  
código.

# Exemplo de representação

Ilustração das chamadas recursivas para calcular 5!



# Exemplo de representação

```
fatorial(6) =  
= 6 * fatorial(5)  
= 6 * (5 * fatorial(4))  
= 6 * (5 * (4 * fatorial(3)))  
= 6 * (5 * (4 * (3 * fatorial(2))))  
= 6 * (5 * (4 * (3 * (2 * fatorial(1)))))  
= 6 * (5 * (4 * (3 * (2 * (1 * fatorial(0)))))) CASO BASE  
= 6 * (5 * (4 * (3 * (2 * (1 * 1))))) =  
= 6 * (5 * (4 * (3 * (2 * 1))))  
= 6 * (5 * (4 * (3 * 2)))  
= 6 * (5 * (4 * 6))  
= 6 * (5 * 24)  
= 6 * 120  
= 720
```



# Caso-base e caso recursivo

Devido ao fato de a função recursiva chamar a si mesma, é mais fácil escrevê-la **erroneamente** e acabar em um loop infinito.

Quando você escreve uma função recursiva, deve informar quando a recursão deve parar.

Por isso, toda função recursiva tem duas partes:

- o caso-base
- e o caso recursivo

# Caso-base e caso recursivo

O **caso recursivo** é quando a função chama a si mesma. Quando chamamos a função dentro dela mesma, devemos sempre mudar o valor do parâmetro passado, de forma que a recursão chegue a um término. Se o valor do parâmetro for sempre o mesmo, a função continuará executando até **esgotar a memória do computador**

O **caso-base** é quando a função **não** chama a si mesma novamente, de forma que o programa não se torna um *loop* infinito.

- O caso-base é o critério de parada, ou seja, o final da recursão. A partir dele volta-se a cada uma das chamadas até a chamada inicial.

No caso do fatorial:

- a chamada recursiva está na linha 5 (slide 8), o retorno ocorrerá apenas quando a multiplicação for efetuada, ou seja, quando a chamada **fatorialRecursivo** ( $n-1$ ), que deu início a uma nova recursão, tiver retornado com o valor calculado para  $N-1$ .
- o caso-base é quando  $n$  é igual 0 e retorna 1.

# Pilha de chamadas recursivas

É uma pilha que armazena informações sobre as sub-rotinas ativas num programa de computador.

Seu principal uso é registrar o ponto em que cada sub-rotina ativa deve retornar o controle de execução quando termina de executar.

Quando a pilha de chamada usa mais memória do que suporta, ocorre um estouro de pilha.

Em diversas linguagens de programação a pilha de chamada possui uma área limitada de memória, geralmente determinada no início do programa. O resultado do uso excessivo de memória é o estouro, o que geralmente resulta na interrupção do programa

# Algoritmos recursivos

1. Cada chamada recursiva deve ser em uma instância menor do mesmo problema, ou seja, um subproblema menor.
2. As chamadas recursivas precisam em algum ponto alcançar um caso base, que é resolvido sem outra recursão.

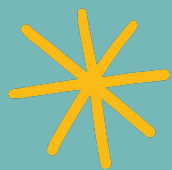


# Melhores práticas para funções recursivas

- definir corretamente o caso base:
  - Esse caso deve indicar quando a recursão deve ser encerrada e retornar um valor concreto.
- divida o problema em subproblemas menores:
  - fundamental para que a função possa se chamar a si mesma de forma adequada
- utilize a recursão de forma eficiente:
  - Evite chamadas recursivas desnecessárias que possam levar a um consumo excessivo de recursos
- faça o uso adequado de parâmetros:
  - cada chamada recursiva deve passar as informações necessárias para a resolução do problema

# Nota

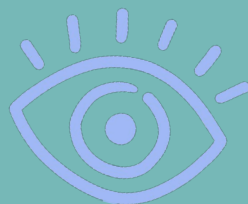
- A recursividade é uma técnica bastante poderosa e muito útil em cenários complexos, empregada com frequência na manipulação de estruturas de dados mais sofisticadas.
- Seu uso demanda diversas considerações sobre complexidade e eficiência de algoritmos (e uso dos recursos computacionais), comparando versões recursivas com suas versões iterativas, entre outras características a serem analisadas.



# Sua vez

?

23



1. Faça uma função recursiva para **somar os números de 1 até n**.

Pense nos valores que n pode assumir.

Qual é o caso-base?

Como deve ser chamada recursiva?

Escreva a pilha de chamadas recursivas



2. Escreva uma função recursiva que calcule a soma dos primeiros  $n$  cubos:

$$S = 1^3 + 2^3 + \dots + n^3$$

3. Crie uma função recursiva que retorne a soma dos elementos de um vetor de inteiros.

5. Crie um subprograma **recursivo** que receba um número inteiro N e imprima todos os números naturais de 0 até N em ordem crescente.

**Entradas:**

Um número inteiro

**Saídas:**

Uma sequência de números naturais de 0 até N

**Exemplo de Entradas:**

15

**Exemplo de Saída:**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

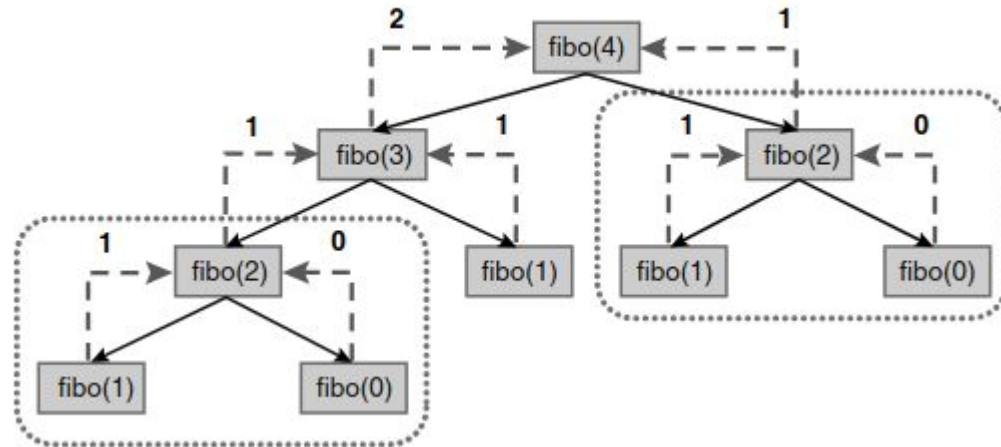
4. Faça uma função recursiva para calcular a potência e retorna o valor de  $x$  elevado a  $n$ .

# Fibonacci

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula a seguir:

Faça a implementação recursiva de Fibonacci. Observe que a função chama a si mesma 2x.



## Sem recursão

```
int fibo(int n){
    int i,t,c,a=0, b=1;
    for(i=0;i<n;i++){
        c = a + b;
        a = b;
        b = c;
    }
    return a;
}
```

# Fibonacci

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula a seguir:

Faça a implementação recursiva de Fibonacci. Observe que a função chama a si mesma 2x.

**Exemplo de entrada:**

7

**Exemplo de saída:**

0

1

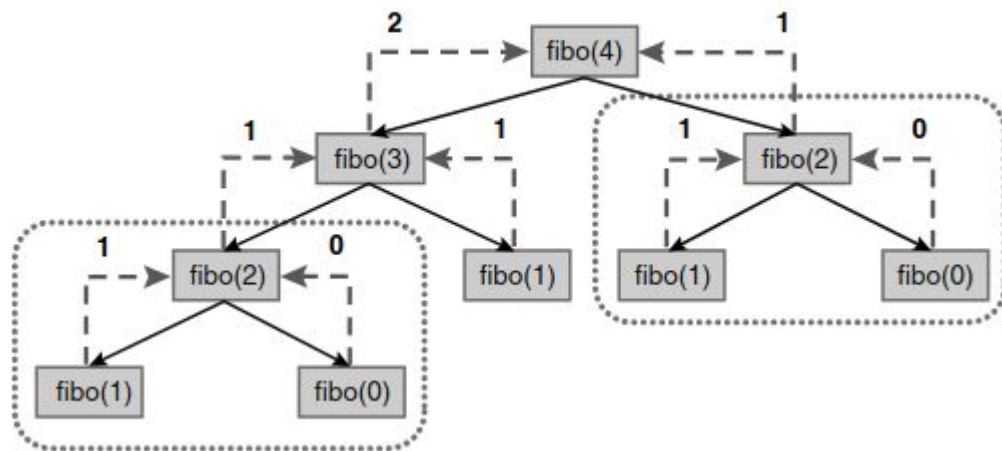
1

2

3

5

8



# Imprimindo uma lista de forma recursiva

Qual a ordem em que os dados serão impressos?

Considere a chamada **imprimir([8,5,4,3], 4)**

O que mudaria se tivesse return na linha 21?

```
18
19 def imprimir(lista,n):
20     if n==1:
21         print(lista[n-1])
22     else:
23         pos=n-1
24         imprimir(lista,pos)
25         print(lista[pos])
26
```