

# Introdução à Programação

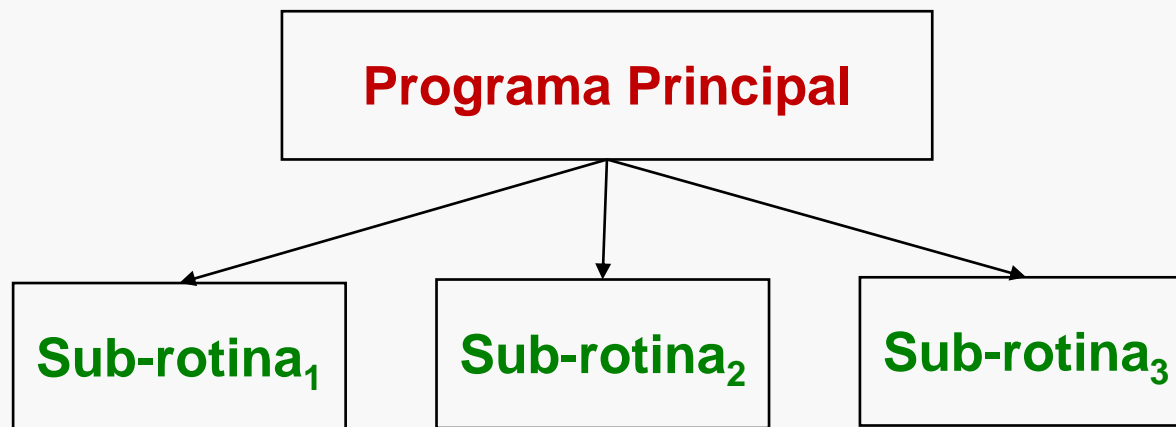
## Aula 14: Modularização – Criando as suas Funções

Prof. Eduardo Corrêa Gonçalves

10/03/2023

# Sub-Rotinas (1/4)

- Modularização é a técnica que consiste em dividir um programa em **partes** que podem conversar umas com as outras.
- Estas partes são denominadas **sub-rotinas**
- **Mais precisamente:** sub-rotina é um bloco de código que pode ser chamado (isto é, executado) a partir de diferentes pontos de um programa.



# Sub-Rotinas (2/4)

- A necessidade de **repetir uma determinada sequência de instruções** muitas vezes em um programa motivou o surgimento das sub-rotinas nas linguagens de programação.
- **Exemplo:** Imagine um programa com 5 matrizes em memória M1, M2, M3, M4 e M5.
  - Suponha que seja necessário imprimir as 5 matrizes.
  - Sem usar uma sub-rotina, você precisa escrever o código abaixo 5 vezes seguidas, trocando apenas o nome da matriz...

```
# imprime a matriz M1
num_lins = len(M1)
num_cols = len(M1[0])
for i in range(num_lins):
    for j in range(num_cols):
        print(f"{M1[i][j]:>10}", end="")
    print()

# e agora tenho que repetir o mesmo código mais 4 vezes p/ M2, M3, M4 e M5
...
```

# Sub-Rotinas (3/4)

- **Exemplo:** Imagine um programa com 5 matrizes em memória M1, M2, M3, M4 e M5.
- Em cada bloco de código repetido:
  - A única coisa que iria mudar é o nome da matriz
  - O resto do código seria todo igual...

```
# imprime a matriz M1
num_lins = len(M1)
num_cols = len(M1[0])
for i in range(num_lins):
    for j in range(num_cols):
        print(f"{M1[i][j]:>10}", end="")
    print()

# e agora tenho que repetir o mesmo código mais 4 vezes p/ M2, M3, M4 e M5
...
```

# Sub-Rotinas (4/4)

- Você aprenderá a trabalhar com sub-rotinas em Python criando as suas próprias **funções**.
- Uma função do Python é uma sub-rotina que pode fazer **duas coisas**:
  - (1). **Computar um valor** e retorná-lo.
    - **Ex.:** retornar o fatorial de um inteiro  $n$ .
  - (2). **Executar uma** determinada ação
    - **Ex.:** imprimir uma matriz.

# Funções – Criação (1/7)

- **Criando Funções**
- A palavra reservada **def** é utilizada para a definição de funções.
- **Exemplo:** programa com definição da função chamada “**faixa\_etaria**”
  - Recebe como **entrada** a **idade** de uma pessoa (número inteiro) e retorna como **saída** a sua **faixa etária** (string).

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'  
  
#chamando a função com diferentes valores  
a = faixa_etaria(15)  
b = faixa_etaria(50)  
c = faixa_etaria(35)  
print(a); print(b); print(c)
```

```
<18  
>=40  
30-39
```

# Funções – Criação (2/7)

- Criação de Funções – Sintaxe
  - Para criar uma função, você deve iniciar escrevendo a palavra **def**



```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'
```

# Funções – Criação (3/7)

- Criação de Funções – Sintaxe
  - Para criar uma função, você deve iniciar escrevendo a palavra **def**
  - Depois indicar o **nome** da função.



```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'
```



# Funções – Criação (4/7)


- Criação de Funções – Sintaxe
  - Para criar uma função, você deve iniciar escrevendo a palavra **def**
  - Depois indicar o nome da função.
  - Depois, entre parênteses, os **parâmetros** que a função deve receber, caso existam.



```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'
```

# Funções – Criação (5/7)

- Criação de Funções – Sintaxe
  - def + nome + parâmetros formam **cabeçalho** da função
  - O cabeçalho deve ser encerrado com dois pontos “:”



```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'
```

**cabeçalho**

# Funções – Criação (6/7)

- Criação de Funções – Sintaxe
  - O **bloco de código** contendo os comandos da função deve ser escrito **alinhado** logo **abaixo do cabeçalho**.
  - O bloco de código também é chamado de **corpo** da função.

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'
```

**bloco de  
código (ou  
corpo) da  
função**

# Funções – Criação (7/7)

- Criação de Funções – Sintaxe

- Dentro do corpo, o comando **return** é usado para retonar um valor.
- Quando um comando return é alcançado:
  - A execução da função termina imediatamente
  - E o valor ao lado do return é retornado para quem chamou a função.

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'
```

**bloco de  
código (ou  
corpo) da  
função**

# Como uma função é executada (1/8)?

- Chamando a função

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'  
  
# chamando a função 3 vezes !!!  
a = faixa_etaria(15)  
b = faixa_etaria(50)  
c = faixa_etaria(35)  
  
print(a)  
print(b)  
print(c)
```

- Quando executarmos esse programa, onde ele iniciará ????
- Ela é a primeira linha do **programa principal**.
- Ou seja, é um código que não está dentro de nenhuma função (e sim dentro do programa principal).

# Como uma função é executada (2/8)?


- Chamando a função

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'  
  
# chamando a função 3 vezes !!!  
a = faixa_etaria(15) ←  
b = faixa_etaria(50)  
c = faixa_etaria(35)  
  
print(a)  
print(b)  
print(c)
```

- Quando executarmos esse programa, onde ele iniciará ????
- Ele iniciará na linha em que está a setinha.
- Ela é a primeira linha do **programa principal**.
  - Ou seja, é um código que não está dentro de nenhuma função (e sim dentro do programa principal).

# Como uma função é executada (3/8)?

- Chamando a função

```
def faixa_etaria(idade): 
    if idade < 18: return '<18'
    elif idade < 30: return '18-29'
    elif idade < 40: return '30-39'
    else: return '>=40'

# chamando a função 3 vezes !!!
a = faixa_etaria(15)
b = faixa_etaria(50)
c = faixa_etaria(35)

print(a)
print(b)
print(c)
```

- Aqui temos a primeira **chamada** (ou **invocação**) da função.
- Como ela é processada pelo Python?
- **a = faixa\_etaria(15)** faz com que "faixa\_etaria" seja executada recebendo como entrada o argumento 15.
- Com isso, o **valor 15** é **automaticamente associado** ao parâmetro "idade" (único parâmetro definido no cabeçalho).

# Como uma função é executada (4/8)?

- Chamando a função

```
def faixa_etaria(idade):  
    if idade < 18: return '<18' ←  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'  
  
# chamando a função 3 vezes !!!  
a = faixa_etaria(15)  
b = faixa_etaria(50)  
c = faixa_etaria(35)  
  
print(a)  
print(b)  
print(c)
```

- Sendo assim, "idade" passa a ter o valor 15 dentro do corpo da função
- Isso faz com que o teste `if (idade < 18)` resulte em `True`
- Por consequência, o comando `return '<18'` é executado.



# Como uma função é executada (5/8)?

- Chamando a função

```
def faixa_etaria(idade):
    if idade < 18: return '<18' ←
    elif idade < 30: return '18-29'
    elif idade < 40: return '30-39'
    else: return '>=40'

# chamando a função 3 vezes !!!
a = faixa_etaria(15)
b = faixa_etaria(50)
c = faixa_etaria(35)

print(a)
print(b)
print(c)
```

- Dentro de uma função, o comando **return** faz com que:
  - a execução da função seja encerrada
  - o valor associado ao comando seja retornado como resultado da função.
- Sendo assim, **return '<18'** **encerra a função e retorna o valor '<18'** para o programa principal.
- Então, **a = faixa\_etaria(15)** faz com que '<18' seja armazenado em "a".

# Como uma função é executada (6/8)?

- Chamando a função

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'  
  
# chamando a função 3 vezes !!!  
a = faixa_etaria(15)  
b = faixa_etaria(50) ←  
c = faixa_etaria(35) ←  
  
print(a)  
print(b)  
print(c)
```

- De maneira análoga, as chamadas:
- `b = faixa_etaria(50)`
- `c = faixa_etaria(35)`
- Fazem com que os valores '`>=40`' e '`30-39`' sejam armazenados na variáveis "b" e "c", respectivamente.

# Como uma função é executada (7/8)?

- Chamando a função

```
def faixa_etaria(idade):  
    if idade < 18: return '<18'  
    elif idade < 30: return '18-29'  
    elif idade < 40: return '30-39'  
    else: return '>=40'  
  
# chamando a função 3 vezes !!!  
a = faixa_etaria(15)  
b = faixa_etaria(50)  
c = faixa_etaria(35)  
  
print(a) ←  
print(b) ←  
print(c) ←
```

- No final deste exemplo, os comandos print do programa principal imprimem os valores de "a", "b" e "c"
- Que foram obtidos através da chamada à função "faixa\_etaria"

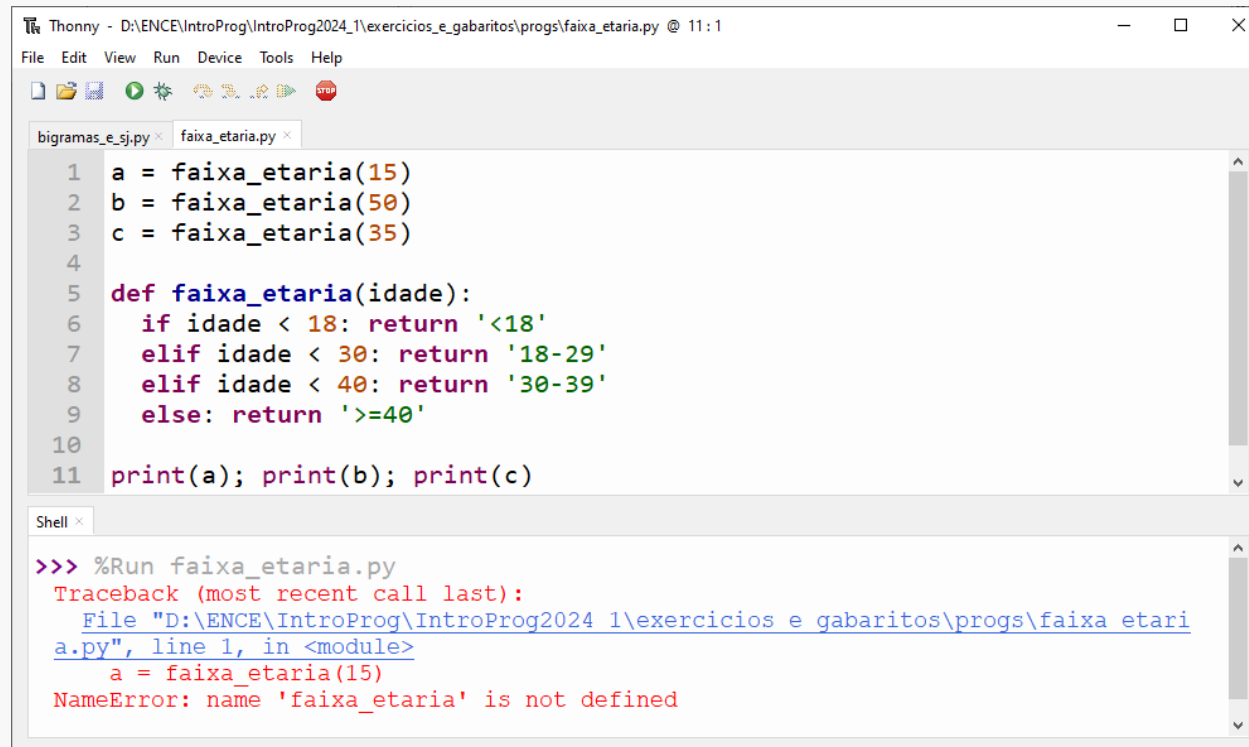
## Saída:

```
<18  
>=40  
30-39
```

# Como uma função é executada (8/8)?

- **IMPORTANTE**

- No programa principal, as funções precisam ser definidas antes de serem chamadas.
- Por este motivo, você não pode, por exemplo, colocar o comando “faixa\_etaria(35)” antes do código que define a função “faixa\_etaria”.
- Veja que a própria IDE Thonny reclamará se você tentar fazer isso.



The screenshot shows the Thonny IDE interface. The top window displays a Python script named `faixa_etaria.py` with the following code:

```
1 a = faixa_etaria(15)
2 b = faixa_etaria(50)
3 c = faixa_etaria(35)
4
5 def faixa_etaria(idade):
6     if idade < 18: return '<18'
7     elif idade < 30: return '18-29'
8     elif idade < 40: return '30-39'
9     else: return '>=40'
10
11 print(a); print(b); print(c)
```

The bottom window, titled "Shell", shows the execution output and an error message:

```
>>> %Run faixa_etaria.py
Traceback (most recent call last):
  File "D:\ENCE\IntroProg\IntroProg2024 1\exercicios e gabaritos\progs\faixa etari
a.py", line 1, in <module>
    a = faixa_etaria(15)
NameError: name 'faixa_etaria' is not defined
```

# Parâmetros (1/4)

- Uma função pode ter qualquer número de parâmetros.
  - Há 2 formas de **passar os argumentos** na chamada da função:
    1. **Posicional**: o significado do argumento é dado por sua posição.
    2. **Keyword**: o significado do argumento é dado por seu nome.
  - **Exemplo 1 – passagem de parâmetro estilo posicional:**

```
def minha_funcao(a, b, c):  
    print(a, b, c)
```

```
minha_funcao(1, 2, 3)
```

*# nesse caso: 1 é associado ao parâmetro “a”, 2 ao “b”, 3 ao “c”  
# a ordem a, b, c será sempre fixa!*

# Parâmetros (2/4)

- Exemplo 2 – passagem de parâmetro estilo keyword:

```
def minha_funcao(a, b, c):  
    print(a, b, c)
```

```
# nesse caso posso usar qualquer ordem  
minha_funcao(a=1, b=2, c=3)  
minha_funcao(b=200, a=100, c=300)
```

- Exemplo 3 – misturando posicional com keyword

```
def minha_funcao(a, b, c):  
    print(a, b, c)
```

```
# você pode misturar os estilos, mas há uma regra a seguir:  
# - Os parâmetros posicionais precisam ser indicados antes dos keyword  
minha_funcao(1, b=2, c=3)    # correto  
minha_funcao(b=200, 1, 3)    # errado
```

# Parâmetros (3/4)

- Parâmetros Opcionais

- A função “soma\_numeros()” possui três parâmetros, mas o terceiro recebe o valor **None** como default.
  - Com isto, ele se torna opcional.

```
def soma_numeros(x, y, z=None) :
    if (z is None) :
        return x+y
    else:
        return x+y+z
```

```
print(soma_numeros(1, 2))
print(soma_numeros(1, 2, 3))
```

```
>>>
3
6
```

# Parâmetros (4/4)

- **Parâmetros com Valor Default**

- Na função “f\_calcula()”, o terceiro parâmetro (“operacao”) possui o valor “+” como default.
- Se a função for chamada sem a especificação deste 3º parâmetro, o valor “+” será automaticamente adotado.

```
def f_calcula(x, y, operacao='+') :  
    if (operacao=='+') : return x+y  
    elif (operacao=='-') : return x-y  
    elif (operacao=='*') : return x*y  
    elif (operacao=='/') : return x/y  
    else: return 'operação inválida!'  
  
print(f_calcula(1, 2)) #retorna 1+2 = 3  
print(f_calcula(1, 2, '+')) #retorna 1+2 = 3  
print(f_calcula(1, 2, '-')) #retorna 1-2 = -1  
print(f_calcula(1, 2, '*')) #retorna 1*2 = 2  
print(f_calcula(1, 2, '/')) #retorna 1/2 = 0.5  
print(f_calcula(1, 2, '.')) #retorna 'operação inválida'
```



# Função sem return (não retorna valor)

- É possível criar uma função que não retorna valor.
  - Basta não usar return.
  - Em geral, é uma função que não vai computar nada.
  - Na verdade ela irá **executar alguma ação** (ex.: imprimir uma matriz)

```

Thonny - D:\ENCE\IntroProg\IntroProg2024_1\exercicios_e_gabaritos\progs\imprime_matriz.py @ 7:9
File Edit View Run Device Tools Help

bigramas_e_sj.py x imprime_matriz.py x

1 def imprime_matriz(mat):
2     m = len(mat)
3     n = len(mat[0])
4     for i in range(m):
5         for j in range(n):
6             print(f"{mat[i][j]:>10}", end="")
7             print()
8
9
10 A = [[100, 200, 300], [400, 500, 600]]
11 imprime_matriz(A)
12

Shell x

>>> %Run imprime_matriz.py

        100         200         300
        400         500         600

>>>

```

# Escopo de Variáveis (1/13)

- **Variável Global *versus* Variável Local**
  - Uma variável é chamada de **global** quando **pertence ao programa principal**,
    - Ou seja: quando foi declarada fora de qualquer função.
  - De maneira oposta, uma variável é chamada de **local** a uma função, quando ela é **declarada dentro da definição da função**.
  - Esses dois tipos de variável “vive” em áreas (ou escopos, no jargão da computação) diferentes dentro de um programa.

# Escopo de Variáveis (2/13)

## • Variável Local

- Uma variável local é visível (existe) apenas dentro do corpo da função a qual ela pertence.
- Ou seja: visível apenas no local onde foi declarada (daí o nome “local”).
- Se você tentar acessá-la fora do corpo da função, ocorrerá um erro, como demonstra o exemplo abaixo.

```
def mostra_soma(a, b):
    z = a + b  # z é uma variável local à função mostra_soma()
    print('SOMA =', z)

mostra_soma(1, 2)
print(z)
```

```
>
SOMA = 3.0
Traceback (most recent call last):
  File "D:\ENCE\erro_variavel_local.py", line 13, in <module>
    print(z)
NameError: name 'z' is not defined
```

# Escopo de Variáveis (3/13)

## • Variável Local

- A variável “z”, é local à função. Isto significa que ela só existe dentro desta função, que só pode ser enxergada pela função “mostra\_soma”.
- Ela nasce quando a função é chamada e morre logo que ela acaba de ser executada! Nem o programa principal e nem qualquer outra função diferente de mostra\_soma são capazes de enxergar “z”!!!

```
def mostra_soma(a, b):
    z = a + b  # z é uma variável local à função mostra_soma()
    print('SOMA =', z)

mostra_soma(1, 2)
print(z)
```

```
>
SOMA = 3.0
Traceback (most recent call last):
  File "D:\ENCE\erro_variavel_local.py", line 13, in <module>
    print(z)
NameError: name 'z' is not defined
```

# Escopo de Variáveis (4/13)

- **Variável Local**

- Veja que colocamos **print(z)** dentro do corpo do programa principal (última linha).
- Ao tentar executar a linha, ocorre o erro **NameError: name 'z' is not defined**.
- Ele ocorre porque o programa principal não conhece nenhum nome “z”, ele é conhecido apenas pela função `mostra_soma`. **Em resumo:** uma variável local nunca poderá ser enxergada fora da função em que foi definida.

```
def mostra_soma(a, b):  
    z = a + b  # z é uma variável local à função mostra_soma()  
    print('SOMA =', z)  
  
mostra_soma(1, 2)  
print(z)
```

```
>  
SOMA = 3.0  
Traceback (most recent call last):  
  File "D:\ENCE\erro_variavel_local.py", line 13, in <module>  
    print(z)  
NameError: name 'z' is not defined
```

# Escopo de Variáveis (5/13)

- **Variável Global (CASO 1: leitura)**
  - Uma variável global é aquela declarada no programa principal.
  - Este tipo de variável pode ter o seu conteúdo enxergado (lido) dentro de qualquer função

```
def mostra_soma():  
    z = a + b  # z é uma variável local à função mostra_soma()  
    print('a =', a) # a é global, e pode ser lida dentro da função  
    print('b =', b) # b é global, e pode ser lida dentro da função  
    print('SOMA =', z)  
  
a = 1  
b = 2  
mostra_soma()
```

```
>  
  
a = 1  
b = 2  
SOMA = 3
```

# Escopo de Variáveis (6/13)

- **Variável Global (CASO 2: modificação)**
  - Porém, se você tentar modificar o conteúdo de uma variável global dentro de um programa a alteração não será refletida para o programa principal.
  - Dá para fazer isso só se você usar um comando chamado **global**, que nem iremos mostrar, pois ele não é muito usado e nem costuma ser recomendado.

```
def f_teste():  
    a = 10  
    print('dentro da função, "a" vale', a)  
  
a = 1  
f_teste()  
print('fora da função, "a" vale', a)
```

>

```
dentro da função, "a" vale 10  
fora da função, "a" vale 1
```

# Escopo de Variáveis (7/13)

- **Variável Global (CASO 2: modificação)**
  - Por que isso acontece?
    - Quando você faz `a = 10` (uma atribuição) dentro do corpo da função... o Python **cria uma variável local chamada "a" !!!**
    - **Ou seja:** passa a existir o "a" local da função e o "a" global do programa principal.
    - E aí, dentro de `f_teste()`, o "a" usado será o local. Fora de `f_teste()`, o "a" usado será o global.

```
def f_teste():  
    a = 10  
    print('dentro da função, "a" vale', a)  
  
a = 1  
f_teste()  
print('fora da função, "a" vale', a)
```

>

```
dentro da função, "a" vale 10  
fora da função, "a" vale 1
```



# Escopo de Variáveis (8/13)

- Alterando uma variável global
  - Considere uma variável  $x$  passada como argumento de uma função:
    - Acabamos de ver que qualquer comando de atribuição envolvendo  $x$  no corpo da função **não será refletido** para fora da função.
    - Isso porque a atribuição “mata” o parâmetro dentro da função, pois o Python “pensa” que você está declarando uma variável local com o mesmo nome!!

```
def soma_um(numero) :  
    numero=numero+1 # agora o Python pensa que numero  
                    # é variável local  
    print("somei um dentro da função: ", numero)  
  
k=100  
print('valor original de k:', k)  
soma_um(k)  
print('Terminou a função e k, na verdade, não mudou:', k)
```

```
valor original de k: 100  
somei um dentro da função: 101  
Terminou a função e k, na verdade, não mudou: 100
```

# Escopo de Variáveis (9/13)

- **Alterando uma variável global**
  - Mas e se você quiser refletir o valor para fora da função. Como fazer?
  - É simples... faça a função retornar um valor!!!

```
def soma_um(numero) :  
    numero=numero+1  
    print("somei um dentro da função: ", numero)  
    return numero  
  
k=100  
print('valor original de k:', k)  
k = soma_um(k)  
print('Terminou a função e agora k mudou:', k)
```

```
valor original de k: 100  
somei um dentro da função: 101  
Terminou a função e agora k mudou: 101
```

# Escopo de Variáveis (10/13)

- Mas dá para alterar lista, dicionário e qualquer outro mutável
  - Não dá para alterar um variável global que seja de um **tipo básico** dentro da função (int, bool, float, string).
  - Na verdade **não dá para alterar** nada que seja **imutável**.
  - Porém, se uma **coleção mutável** (como uma lista ou dicionário) é passada como argumento de uma função, ela **pode ser alterada !!**
  - Veja o exemplo da página a seguir.

# Escopo de Variáveis (11/13)

- Mas dá para alterar lista, dicionário e qualquer outro mutável

```
# passagem de parâmetro - variável do tipo coleção
def f_dummy(lista):
    lista.append(999) # você pode modificar a lista usando métodos
    lista[0] = -1     # você pode modificar elementos específicos

lst = [1,2,3,4,5]
print('lst antes de chamar a função f_dummy:', lst)

f_dummy(lst)
print('lst depois de chamar a função f_dummy:', lst)
```

```
>>>
```

```
lst antes de chamar a função f_dummy: [1, 2, 3, 4, 5]
```

```
lst depois de chamar a função f_dummy: [-1, 2, 3, 4, 5, 999]
```

# Escopo de Variáveis (12/13)

- **MAS VOCÊ DEVE TOMAR UM ÚNICO CUIDADO**
  - Você deve apenas tomar um único cuidado: dentro do corpo da função, você não pode recriar a lista usando uma operação de atribuição.
  - Se você fizer isso, o Python vai “matar” o seu parâmetro e vai criar uma variável local com o mesmo nome, fazendo com que qualquer alteração deixe de ser refletida para o programa principal.
  - Compare o exemplo do próximo slide com o exemplo anterior.

# Escopo de Variáveis (13/13)

- Mas dá para alterar lista, dicionário e qualquer outro mutável

```
# passagem de parâmetro - não faça uma atribuição
def f_dummy(lista):
    lista = [10, 20, 30] # você não pode recriar a lista
                        # dentro da função (ela vira variável local)

lst = [1,2,3,4,5]
print('lst antes de chamar a função f_dummy:', lst)

f_dummy(lst)
print('lst depois de chamar a função f_dummy:', lst)
```

>

```
lst antes de chamar a função f_dummy: [1, 2, 3, 4, 5]
lst depois de chamar a função f_dummy: [1, 2, 3, 4, 5]
```

# \*args – um recurso interessante

- **\*args**: recurso que nos permite passar um **número arbitrário de parâmetros** para uma função

```
def pessoa(nome, *args):  
    print("- nome (primeiro parâmetro): ", nome)  
    print("- características (outros parâmetros): ")  
    for arg in args:  
        print("\t", arg)  
  
pessoa('Jane', 'escritora', 'sagitariana', 'romântica')  
pessoa('John', 'músico')
```

- nome (primeiro parâmetro): Jane
- características (outros parâmetros):  
escritora  
sagitariana  
romântica
- nome (primeiro parâmetro): John
- características (outros parâmetros):  
músico

# Funções lambda

- **lambda**
  - Notação abreviada que pode ser empregada para definir funções simples (uma expressão, precisa ser uma única linha)
  - Muito popular entre os *pythonistas*!
  - Sintaxe: **lambda** parâmetros: expressão.

```
f_etaria = lambda idade: 'menor' if idade < 18 else 'maior'

print(f_etaria(20))    #maior
print(f_etaria(16))    #menor
```



# Exercícios

- (1) Crie uma função que receba como entrada um valor de temperatura em graus Celsius (C) e que retorne o valor equivalente em graus Fahrenheit (F) através da seguinte fórmula:

$$F = 1,8 \times C + 32.$$

- (2) Sejam  $P(x_1, y_1)$  e  $Q(x_2, y_2)$  dois pontos quaisquer do plano. A distância entre eles é dada por:  $d = \text{raiz}((x_2 - x_1)^2 + (y_2 - y_1)^2)$ .
  - Crie uma função que receba como entrada os valores de  $x_1$ ,  $y_1$ ,  $x_2$  e  $y_2$ , representando as coordenadas de dois pontos, e que compute como saída a distância entre estes pontos.
- (3) – A Sequência de Fibonacci tem como primeiros termos os números 0 e 1 e, a seguir, cada termo subsequente é obtido pela soma dos dois termos predecessores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

- Crie uma função “Fibonacci( $n$ )” que retorne como saída uma lista contendo os  $n$  primeiros números da sequência de Fibonacci. Utilize essa função em um programa em que o usuário digite o valor de  $n$  para, em seguida, ver a sequência. O programa deve ser mantido em execução enquanto o usuário desejar.