

CC8210 – NCA210

Programação Avançada I

Prof. Reinaldo A. C. Bianchi

Prof. Isaac Jesus da Silva

Prof. Danilo H. Perico

Conteúdo Programático

AULA	DATA	TEORIA
1	10/08/20	Introdução a disciplina. Introdução a Python :Estrutura Sequencial, Condicional e Estruturas de Repetição;
2	17/08/20	Vetores ou Listas;
3	24/08/20	Listas Aninhadas ou Matrizes;
4	31/08/20	Modularização (Funções com/sem passagem de parâmetro, com/sem retorno);
5	14/09/20	Manipulação de Strings
6	21/09/20	Manipulação de Arquivos
7	28/09/20	Dicionários e Tuplas
	05/10/20	P1
8	19/10/20	Introdução à POO - Classes e Objetos
9	26/10/20	Introdução à biblioteca Pandas
10	09/11/20	Introdução à bibliotecas matemáticas e gráficas numpy e matplotlib. Manipulação de gráficos múltiplos e de barras;
11	16/11/20	Interface Gráfica do Usuário (GUI)
12	23/11/20	P2

Mudularização: Funções

Funções ou Subprogramas

- A construção de funções, também chamados de subprogramas é o recurso disponível nas linguagens de programação para elaborar-se a abstração de processos.
- São blocos de construção fundamentais dos programas.
 - Aumenta a legibilidade e facilita a manutenção;
 - Reduz o tempo de codificação e o uso de memória.

Subprogramas

- Definição:
- São unidades de programação que contêm uma seqüência de instruções que, se for executada, cumpre algum objetivo bem específico e determinado.
- Permitem reutilização do código que executa uma operação, escondendo detalhes de implementação e favorecendo a abstração.

Subprogramas

- Funções são blocos de código que realizam determinadas tarefas que normalmente precisam ser executadas diversas vezes dentro da mesma aplicação
- Assim, tarefas muito utilizadas costumam ser agrupadas em funções, que, depois de definidas, podem ser utilizadas / chamadas em qualquer parte do código somente pelo seu nome

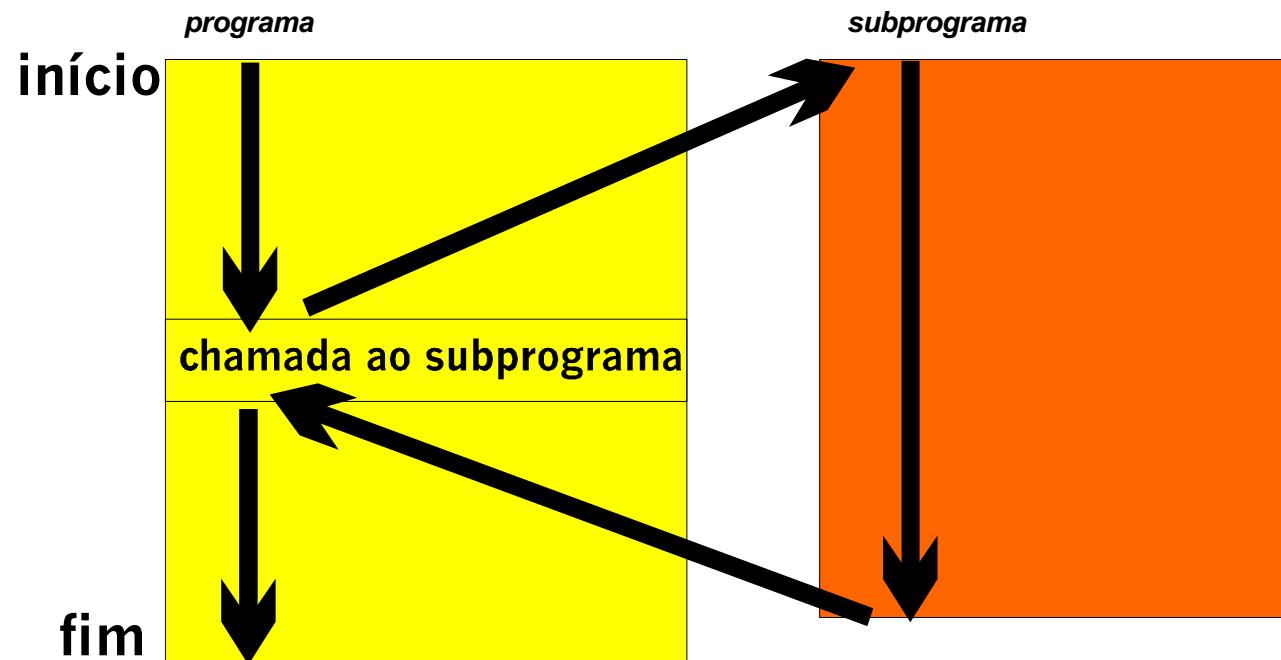
Subprogramas

- São conhecidos nas linguagens mais antigas como subrotinas:
 - Fortran, Assembly
- São conhecidas nas linguagens procedimentais como funções e procedimentos:
 - C, Pascal, Python
- Nas linguagens orientadas a objetos os subprogramas são chamados de métodos e estão encapsulados nas classes dos objetos.

Subprogramas: características fundamentais

- Todos os subprogramas têm as seguintes características:
- Cada subprograma tem um único ponto de entrada.
- A unidade chamadora é suspensa durante a execução do subprograma: somente um subprograma se encontra em execução em um determinado instante.
- O controle volta ao chamador quando o subprograma encerra-se.

Subprogramas: características fundamentais



Subprogramas: Definições Básicas

- A definição de subprograma descreve a interface e as ações da abstração de subprograma.
 - é composta pelo identificador desse subprograma.
 - pela indicação de seus parâmetros, que constituem interface dessa unidade.
 - pela descrição das ações que são executadas pelo subprograma (corpo do subprograma).

Subprogramas: Chamadas

- Uma chamada a subprograma é a solicitação explícita para executar o subprograma.
- O Subprograma se encontra ativo se depois de ter sido chamado, iniciou a execução e não a terminou.
- Existem duas categorias distintas de subprogramas: função ou procedimento.

Subprogramas: Cabeçalho

- O cabeçalho de subprograma é a primeira linha de sua definição.
- Tem diversos propósitos:
 - indica que aquela unidade sintática é a definição de um subprograma.
 - especifica o identificador desse subprograma.
 - especificar a lista (opcional) de parâmetros do subprograma.

Definindo funções em Python - **def**

- Podemos criar / definir nossas próprias funções no Python utilizando a palavra-chave def seguido do nome da função, parêntesis () e ":"
- Sintaxe:

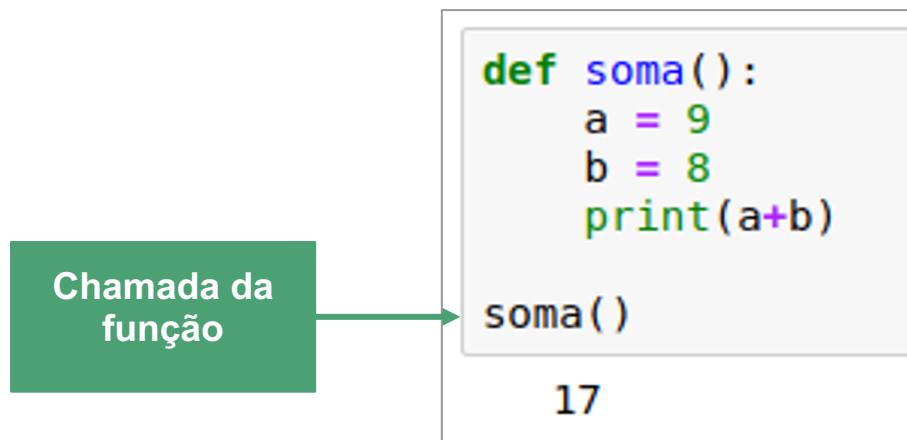
```
def <nome da função>():  
    # tarefas que serão realizadas dentro da função
```

- Exemplo:

```
def imprimeOlá():  
    print("Olá")
```

Definindo funções em Python - **def**

- Exemplo:



Rules for naming Python function (identifier)

- We follow the same rules when naming a function as we do when naming a variable.
 - It can begin with either of the following: A-Z, a-z, and underscore(_).
 - The rest of it can contain either of the following: A-Z, a-z, digits(0-9), and underscore(_).
- A reserved keyword may not be chosen as an identifier.
- It is good practice to name a Python function according to what it does.

Parâmetros

- Os parâmetros de um subprograma constituem um meio que permite o trânsito (“troca”) de dados entre o subprograma e a unidade chamadora.
- É sempre mais interessante estabelecer o trânsito de dados exclusivamente por meio dos parâmetros, pois acarreta uma característica importante que é a independência do subprograma em relação ao programa onde é utilizado.

Subprogramas: Parâmetros

- O perfil de parâmetro de um subprograma indica a quantidade, a ordem e os tipos de seus parâmetros formais.
- O protocolo de um subprograma é o seu perfil de parâmetro e mais o tipo de retorno que fornece - se for uma função.

Parâmetros posicionais

- Na maioria das linguagens, a correspondência entre os parâmetros de entrada e as variáveis do programa chamador é estabelecida simplesmente pela ordem seqüencial :
 - primeiro com primeiro,
 - segundo com segundo ...
- Nessa forma de correspondência são chamados parâmetros posicionais.

Parâmetros nomeados

- Algumas linguagens permitem ainda estabelecer a correspondência a partir dos nomes dos parâmetros. São os parâmetros nomeados. Exemplo:
 - `SORT(LIST => A, LENGTH => N);`
- Vantagem: a ordem é irrelevante.
- Desvantagem: o usuário precisa saber o nome do parâmetro formal.

Nota

- Em Python os parâmetros também são chamados de argumentos.
 - Referencia à matemática

Uso de parâmetros

- As funções podem ou não ter **parâmetros**, que **são valores enviados às funções** dentro dos parêntesis no momento em que elas são chamadas
- Exemplos:

Sem parâmetros:

```
def soma():
    a = 9
    b = 8
    print(a+b)
```

17

Chamada
da função

Com parâmetros:

```
def soma(a, b):
    print(a+b)
```

7

Chamada
da função

Procedimentos x Funções

- Um procedimento representa um conjunto de instruções que define uma computação parametrizada. Os resultados na unidade chamadora são produzidos de duas maneiras:
 - Uso de variáveis globais
 - Execução de uma ação, como limpar a tela
 - Uso de parâmetros formais como referência às variáveis do programa.
- **Não retornam valores!**

Procedimentos x Funções

- Uma função lembram os procedimentos mas seguem o modelo de funções matemáticas.
- Uma função não deve alterar qualquer um dos parâmetros reais incluídos na respectiva instrução de chamada nem variáveis globais.
- O efeito da execução de uma função deve ser exclusivamente o retorno de um dado, ou seja: **Retornam valores!**

Procedimentos x Funções

- Procedimentos:

- Em Pascal:

```
Procedure fracao(Var x,y:integer);
```

- Em C:

```
void fracao(int &x, int &y);
```

- Funções:

- Em Pascal:

```
Function max(x,y:integer):integer;
```

- Em C: int max(int x, int y);

Nota

- Em Python, C, Java, não se utiliza mais o nome procedimento com frequência...
- Em Python, a declaração é sempre a mesma, visto que não de declara o tipo de retorno, nem se existirá um retorno no cabeçalho da função.

Funções - *return*

- As funções podem ou não ter um **valor de retorno**
- O retorno é definido pela palavra-chave ***return***
- Exemplos:

Sem parâmetros:

```
def soma():
    a = 9
    b = 8
    return(a+b)

print(soma())
```

17

Com parâmetros:

```
def soma(a, b):
    return(a+b)

print(soma(3,4))
```

7

Funções

- Exemplo: Fazer uma função que retorne ***True*** ou ***False*** para a verificação de números pares.
 - Precisa de parâmetros? Sim ou Não?
 - É melhor usar ou não o ***return***?

Funções

- Exemplo: Fazer uma função que retorne *True* ou *False* para a verificação de números pares.

```
def par(num):  
    return(num % 2 == 0)
```

```
print(par(3))  
print(par(4))  
print(par(67))
```

False

True

False

Funções

- Exemplo:
- Se precisarmos de uma função que retorne a string “*par*” ou “*ímpar*”, podemos reutilizar a função *par* e criar uma função nova:

```
def par(num):  
    return(num % 2 == 0)  
  
def parOuImpar(x):  
    if par(x) == True:  
        return "par!"  
    else:  
        return "ímpar!"  
  
print(parOuImpar(4))  
print(parOuImpar(3))  
print(parOuImpar(56))
```

par!
ímpar!
par!

Funções

- A grande diferença de Python para outras linguagens de programação:
- Não é necessário definir os tipos dos parâmetros nem do que a função retorna!
- Exemplo em C:

```
int quadrado (int x)
{
    return (x * x);
}
```

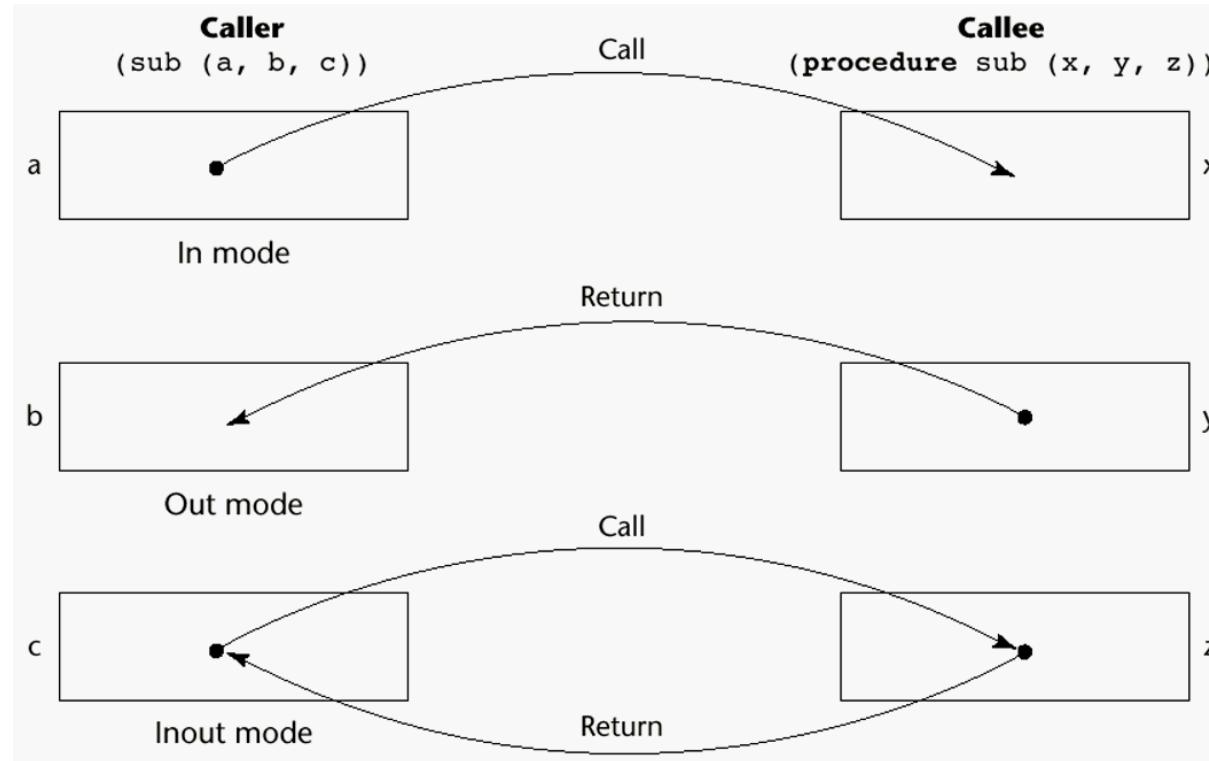
Passagem de Parâmetros: mais detalhadamente

- Os Métodos de Passagem de Parâmetros são a maneira pelas quais se transmitem parâmetros dos programas aos subprogramas chamados.
- Podem ser analisados sob dois aspectos:
 - Semântico
 - Implementação.

Passagem de Parâmetros: modelos semânticos

- Há três modelos semânticos fundamentais de passagem de parâmetros:
 - modo entrada (*in mode*): o parâmetro formal pode receber o dado do parâmetro real.
 - modo saída (*out mode*): o parâmetro formal pode transmitir o dado ao parâmetro real.
 - modo entrada-saída (*in-out mode*): o parâmetro formal pode receber ou transmitir o dado.

Passagem de Parâmetros: modelos semânticos



Passagem de Parâmetros: modelos de implementação

- Há cinco modelos de implementação fundamentais de passagem de parâmetros:
 - por valor.
 - por resultado.
 - por valor-resultado.
 - por referência.
 - por nome.

Passagem de Parâmetros por Valor - modo entrada

- O valor do parâmetro real é usado para inicializar o parâmetro formal que se configura como uma variável local ao subprograma.
- Nessa situação, o parâmetro real pode ser uma expressão cujo valor ou resultado será transmitido ao subprograma, inicializando o parâmetro formal correspondente.

Passagem de Parâmetros por Resultado - modo saída

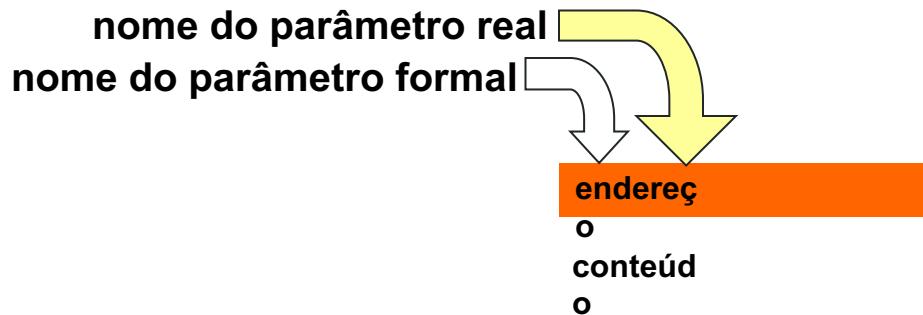
- O parâmetro formal, que se configura como uma variável local ao subprograma, tem o seu conteúdo transmitido ao parâmetro real imediatamente antes da execução (do subprograma) ser encerrada.
- Nessa situação, o parâmetro real deve ser uma variável cujo conteúdo será redefinido como efeito da execução do subprograma.
 - É o nosso famoso `return`

Passagem de Parâmetros por Referência - modo in-out

- As referências, no corpo do subprograma, feitas ao parâmetro formal, efetivamente são referências ao parâmetro real.
- Assim, qualquer operação de redefinição de conteúdo do parâmetro formal resulta, efetivamente, em redefinição do conteúdo do parâmetro real.

Passagem de Parâmetros por Referência - modo in-out

*os nomes dos parâmetros (real e formal)
são aliases, identificam a mesma variável.*



Toda alteração feita no parâmetro formal reflete-se no parâmetro real porque a posição de memória, referida pelo endereço, é a mesma.

Passagem de Parâmetros: Comparação

- Por valor: as operações de transferência e armazenamento podem ser custosas.
- Por resultado: podem ocorrer problemas de “colisão” de parâmetros reais.
- Por valor-resultado: Ambos acima.
- Por referência:
 - É necessário um nível a mais de endereçamento (acesso mais lento);
 - Não há duplicação de armazenamento nem tarefa de cópia.

Passagem de Parâmetros em Python

- O modelo de passagem de parâmetros em Python é a “passagem por referência de objeto”
- Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing"
 - Object references are passed by value.

Passagem de Parâmetros em Python

- Todos os parâmetros (argumentos) na linguagem Python são passados por referência.
- Significa que se você alterar o que um parâmetro se refere dentro de uma função, a mudança também reflete de volta na função de chamada.

Passagem de Parâmetros em Python

- No caso de você passar argumentos como números inteiros, strings ou tuplas para uma função, a passagem é como chamada por valor porque você não pode alterar o valor dos objetos imutáveis que estão sendo passados para a função.

PythonTutor.com - passagem por valor

```
string = "Geeks"  
def test(string):  
    string = "GeeksforGeeks"  
    print("Inside Function:", string)  
# Driver's code  
test(string)  
print("Outside Function:", string)
```

PythonTutor.com - passagem por referência

```
def add_more(list):
    list.append(50)
    print("Inside Function", list)

# Driver's code

mylist = [10,20,30,40]
add_more(mylist)

print("Outside Function:", mylist)
```

Ambientes de referência locais

- Subprogramas podem definir suas próprias variáveis, definindo os chamados ambientes de referência locais.
- São as variáveis locais, cujo acesso é feito só no subprograma.
- Essas variáveis locais podem ser estáticas ou stack-dinâmicas (perdem o conteúdo entre chamadas distintas).

Ambientes de referência locais

- Em FORTAN 77 e 90:
 - a maioria é estática.
- Em C, C++:
 - por default é dinâmica, mas pode ser declarada estática.
- Pascal, Modula-2, and Ada:
 - Somente dinâmicas.
- Python:
 - Like most other languages, Python is *statically* scoped

Escopo das variáveis: **locais**

- Quando usamos funções, trabalhamos com variáveis internas, que pertencem somente ao ambiente de referência local da função:
 - Estas variáveis internas são chamadas variáveis locais
- Não podemos acessar os valores das variáveis locais fora da função a que elas pertencem.
- É por isso que passamos parâmetros e retornamos valores das funções.
 - Os parâmetros e o return possibilitam a troca de dados no programa

Escopo das variáveis: **globais**

- Por sua vez, as **variáveis globais** são **definidas fora das funções** e podem ser vistas e acessadas por todas as funções e pelo “*código principal*” (que não está dentro de uma função específica)

Escopo das variáveis: locais *vs.* globais

```
# variável global:  
a = 5  
  
def alteraValor():  
    # variável local da função alteraValor():  
    a = 7  
    print("Dentro da função 'a' vale: ", a)  
  
    print("'a' antes da chamada da função: ", a)  
alteraValor()  
print("'a' depois da chamada da função", a)
```

```
'a' antes da chamada da função: 5  
Dentro da função 'a' vale: 7  
'a' depois da chamada da função 5
```

Escopo das variáveis: locais *vs.* globais

- Se quisermos modificar a variável global dentro da função, devemos utilizar a palavra-chave **global**

```
# variável global:  
a = 5  
  
def alteraValor():  
    # dizemos para a função que a variável 'a' é global:  
    global a  
    a = 7  
    print("Dentro da função 'a' vale: ", a)  
  
print("'a' antes da chamada da função: ", a)  
alteraValor()  
print("'a' depois da chamada da função", a)
```

'a' antes da chamada da função: 5
Dentro da função 'a' vale: 7
'a' depois da chamada da função 7

Execute no PythonTutor.com

```
a = 5
```

```
def alteraValor ():
```

```
    a = 7
```

```
    print ("O valor dentro da funcao : ", a)
```

```
print ("O valor antes da funcao : ", a)
```

```
alteraValor ()
```

```
print ("O valor depois da funcao : ", a)
```

Funções - parâmetros opcionais

- Podemos ainda criar funções que podem ou não receber argumentos.
- Exemplo:

```
def soma(a=1, b=1):  
    print(a+b)  
  
soma()  
soma(2,3)
```

2
5

Chamada sem argumento:
Neste caso, *soma* assume valores 1 para **a** e **b**

Chamada com argumentos:
Neste caso, *soma* assume valores 2 para **a** e 3 para **b**

Funções Lambda

- No Python, podemos criar funções simples em somente uma linha
- Estas funções, ou expressões, são chamadas de lambda
- Exemplo: função lambda que recebe um parâmetro x e retorna o quadrado deste número.
 - lambda cria uma função a

```
a = lambda x: x**2  
print(a(3))
```

9

Funções Lambda

- Exemplo:
- Função lambda que calcula o aumento, dado o valor inicial e a porcentagem de aumento:

```
aumento = lambda a,b : a*b/100
aumento(100,5)
```

5.0

Funções Recursivas

- Uma função recursiva é uma função que se refere a si própria.
 - A ideia consiste em utilizar a própria função que estamos a definir na sua definição.
- A execução de uma função recursiva consiste em ir resolvendo subproblemas sucessivamente mais simples até se atingir o caso mais simples de todos, cujo resultado é imediato.

Funções Recursivas

- Em todas as funções recursivas existe:
 - Um passo básico (ou mais) cujo resultado é imediatamente conhecido.
 - Um passo recursivo em que se tenta resolver um sub-problema do problema inicial.
- Geralmente, uma função recursiva só funciona se tiver uma expressão condicional.

Funções Recursivas

- Desta forma, o padrão mais comum para escrever uma função recursiva é:
 - Começar por testar os casos mais simples.
 - Fazer chamada (ou chamadas) recursivas com subproblemas cada vez mais próximos dos casos mais simples.
- Como exemplo clássico, podemos citar a Função Fatorial, que pode ser declarada como a seguir:
 - $x! = x * (x-1) * (x-2) * \dots * (3) * (2) * (1)$

Fatorial Recursivo em Python

```
def fatorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n-1)
```

```
print(fatorial(5))
```

Conclusão

- Subprogramas = Subrotinas = métodos
- Tipos:
 - Procedimento
 - Função
 - Em Python, não existe diferença formal.
- Passagem de parâmetros é delicada.

Python Functions

