

UFCD 10793-04/AG – Fundamentos em Python

João Araújo

01-03-2024



Cofinanciado por:



Controlo de fluxo



ATENÇÃO!!!!!!

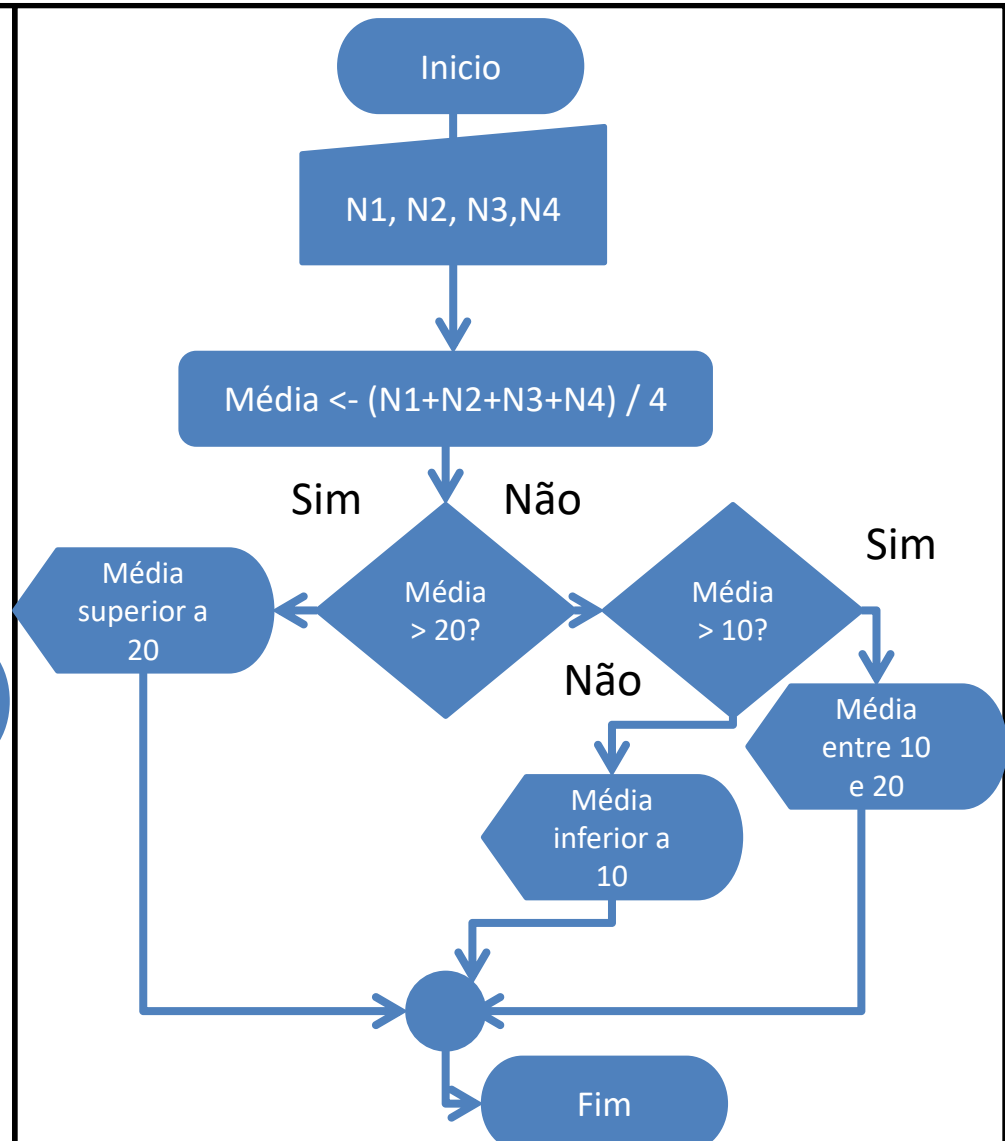
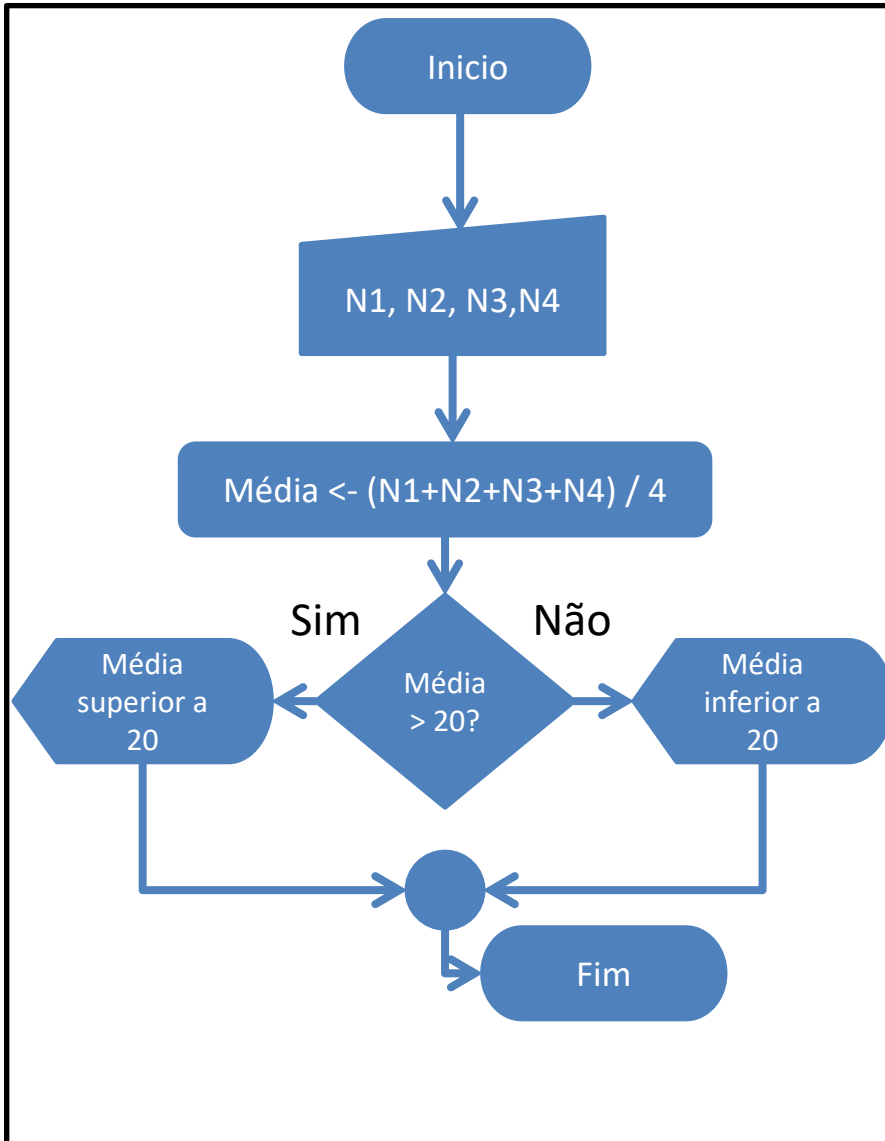
A indentação é crucial para todas as instruções de controlo de fluxo.

Estruturas de seleção

- Condição IF
 - Controlo de uma ou mais condições, através de expressões lógicas. Se a condição se verificar, executa o código associado(if x == 0:)
- Condição IF-ELSE
 - Controlo de uma ou mais condições, através de expressões lógicas. Se a condição se verificar, executa o código associado à parte IF. Se não, executa o código associado em ELSE
 - if (x == 0):
 - print('valor zero')
 - else:
 - print('valor não zero')
- Condição IF-ELIF-ELSE
 - Condição que especifica múltiplas condições e executa a que se verificar
 - if (x == 0):
 - print('valor zero')
 - elif (x == 1)
 - print('valor um')
 - else
 - print('valor diferente de zero e um')
- Condição MATCH-CASE
 - Condição que especifica a condição de uma variável, e executa o código em função do seu conteúdo:
 - match x:
 - case 0:
 - print('Valor zero')
 - case 1:
 - print('Valor 1')
 - case other:
 - print(valor não é 0 nem 1')

Controlo de fluxo

Qual a condição de seleção dos dois algoritmos?

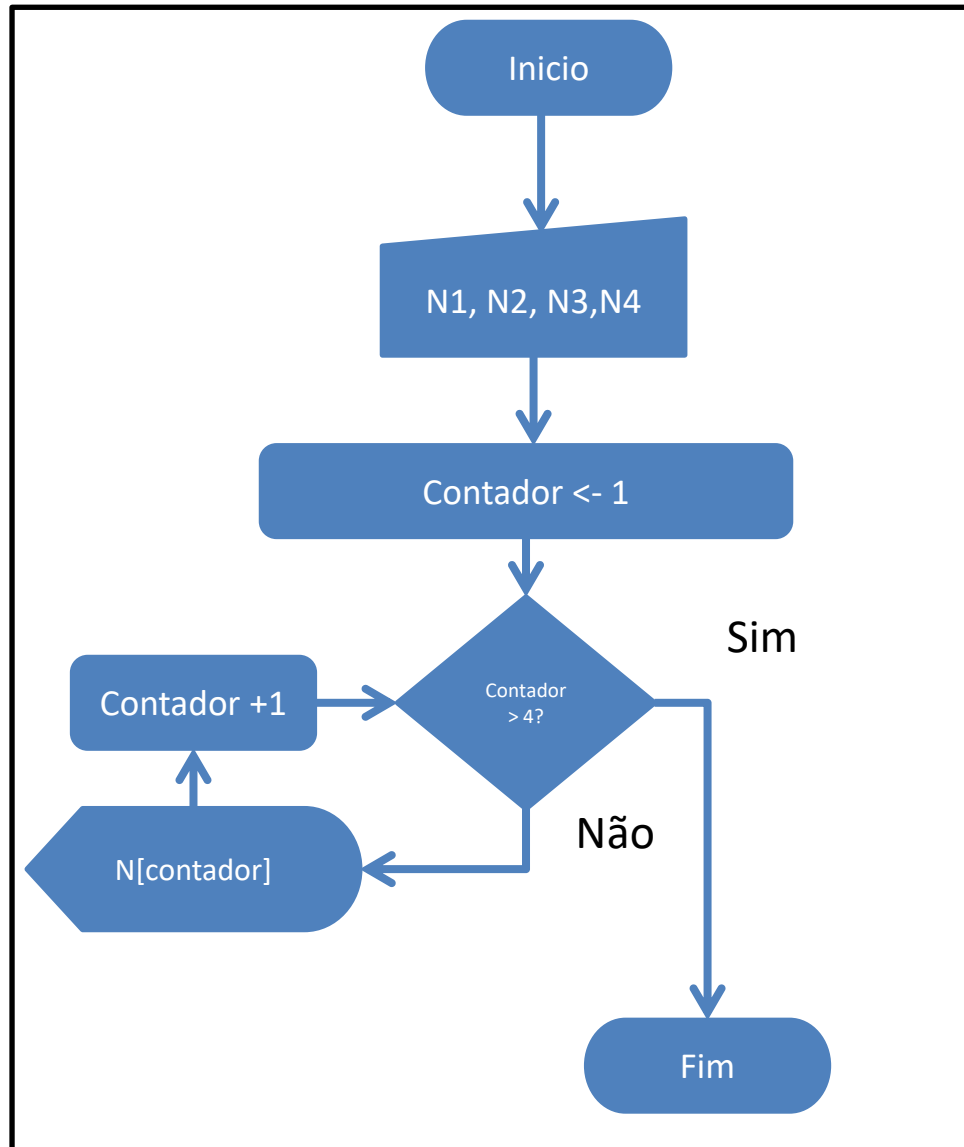


Controlo de fluxo

Estruturas de repetição

- Condição For
 - Faz iterações sobre uma variavel de multiplos dados em sequencia (*list, set, tuples, strings,...*)
 - for x in range(n):
 - print('Iteration' + i)
- Condição While
 - Executa um conjunto de código, deste que a condição seja verdadeira
 - while x < n
 - print(x)

Controlo de fluxo



Controlo de fluxo

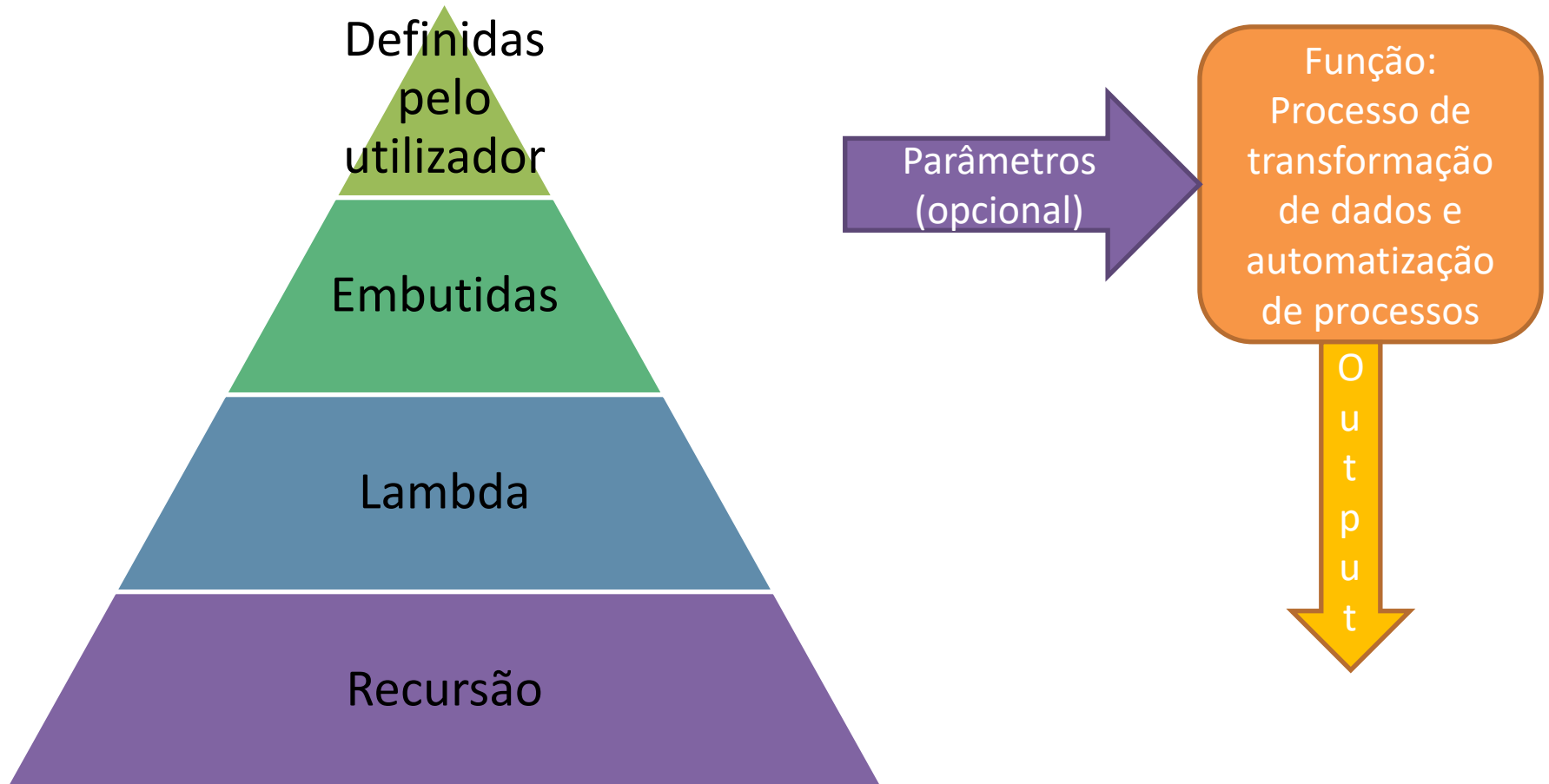
- Iteradores
 - Permite fazer iterações em variáveis iteráveis em qualquer ponto do programa pelo programador
 - Todas as variáveis em que é possível implementar o método `iter()` são iteráveis
 - Quando uma variável fica definido como iterador, pode-se aplicar o método `next()`, para iterar na variável em qualquer ponto do programa

Controlo de fluxo

Tratamento de exceções

- Condição try, except, finally
 - Deteta se existe um erro ao executar uma operação, e como o programa deve agir
 - try:
 - `x = int('a')`
 - except [especificação opcional]:
 - `print('Input inválido')`
 - finally:
 - `print('tarefa terminada')`
- Se não houver especificação de *except*, qualquer erro é executado na sua instrução
- <https://docs.python.org/3/library/exceptions.html>

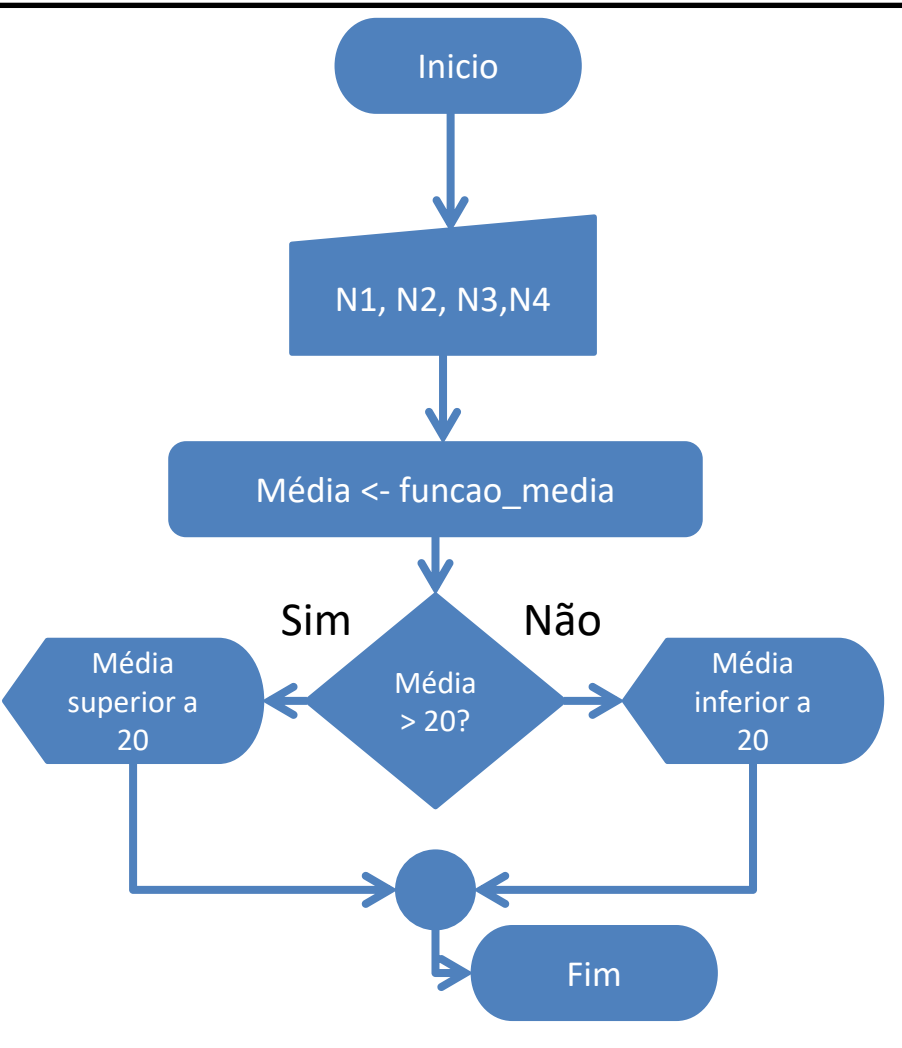
Funções em Python



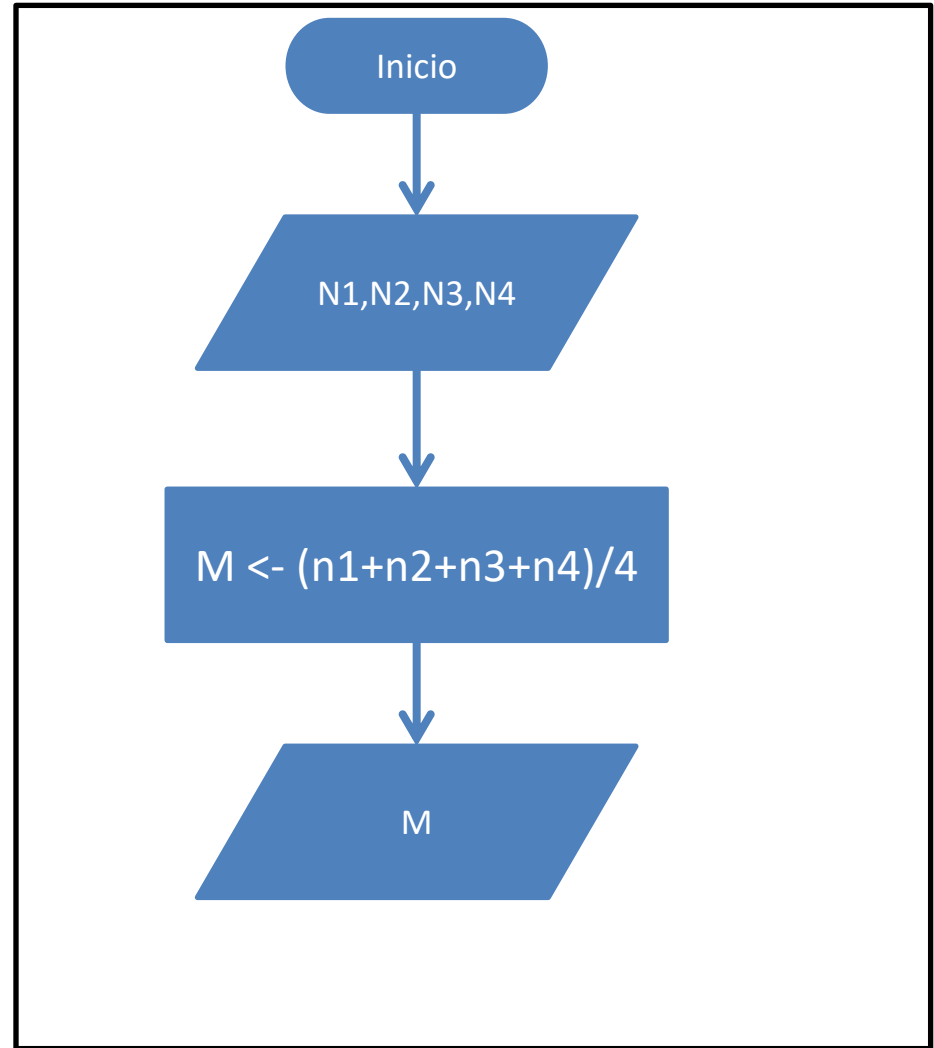
Funções

- Exemplo: Algoritmo de cálculo de uma média

Programa principal



Função média



Funções

- Definição de funções
 - `def myFunc(opt_args):`
 - `#Corpo`
 - `#da`
 - `#função`
 - `return n` (opcional)
 - Permite automatizar processos dentro do programa que são utilizados por várias instâncias
 - Tal como as funções de controlo de fluxo *if, for,...*, terá sempre de se respeitar a indentação do código
 - Opções
 - Parâmetros de entrada
 - Retorno de um ou mais elementos após a execução da função

Funções

- Função procedimental
 - `def myFunc():`
 - `#corpo`
 - `#da`
 - `#função`
- Para executar a função será apenas colocar o seu nome seguido de parênteses no código, depois desta ser definida
 - `myFunc()`
- Argumentos
 - Pode-se passar informação para a execução das funções através de argumentos. Pode-se colocar um número ilimitado de argumentos, separados por vírgulas
 - `def myFunc(arg1, arg2):`
 - `#corpo`
 - `#da`
 - `#função`
 - `arg1`
 - `myFunc(arg1, arg2)`

Funções

- Argumentos arbitrários
 - Se não é possível saber quantos argumentos serão passados na função utiliza-se o seguinte método
 - `def myFunc(*args):`
 - `#corpo`
 - `#da`
 - `#função`
 - `args[1]`
 - A função irá receber um *tuple* de argumentos, de acordo com o que for referenciado ao invocar a função
 - `myFunc("arg1", "arg2", "arg3")`
- Argumentos por palavra-chave
 - É possível usar como argumentos uma variável com elemento que se pretende enviar para a função. Deste modo, a ordem dos argumentos não interessa
 - `def myFunc(arg3,arg2,arg1)`
 - `#corpo`
 - `#da`
 - `#função`
 - `arg1`
 - `myfunc(arg1 = "element1", arg2 = "element2", arg3 = "element3")`

Funções

- Argumentos arbitrários por palavra chave
 - Se não é possível saber quantos argumentos por palavra chave serão passados para a função, utiliza-se o seguinte método
 - `def myFunc(**kwargs):`
 - `# corpo`
 - `# da`
 - `# função`
 - `kwargs["arg1"]`
 - A função irá receber um *dictionary* de elementos, que se envia da seguinte forma
 - `myFunc(arg1 = "element1", arg2 = "element2")`

Funções

- Função *lambda*
 - Pequena função anónima do *Python* (não existe identificador da função)
 - Pode conter qualquer numero de argumentos, mas pode **apenas** conter uma expressão
 - `x = lamda args : #expressão`
 - `x(args)`
 - É possível integrar a função lambda dentro de funções não anónimas
 - `def myFunc(args):`
 - `return lambda l_args : l_args * args`
 - `funcVariable = myFunc(args)`
 - `funcVariable(l_args)`

Funções

- Variáveis globais
 - Variáveis criadas fora de qualquer função no programa
 - Podem ser acedidas por todos os métodos ou expressões
 - `myVariable = variavel`
 - `def myFunc():`
 - `print(myVariable)`
 - `myFunc()`
- Variáveis locais
 - Variáveis criadas dentro de qualquer função no programa
 - Apenas podem ser acedidas dentro das funções onde foram criadas
 - `def myFunc():`
 - `myVariable = variavel`
 - `print(myVariable)`
 - `myFunc()`
- Podem ser criadas variáveis globais dentro de funções, declarando a variável como *global*
 - `def myFunc():`
 - `global variavel`
 - `myVariable = variavel`
 - `print(myVariable)`
 - `myFunc()`
- As variáveis locais podem ser acedidas por funções dentro de funções
 - `def myFunc():`
 - `myVariable = variavel`
 - `def myInFunc()`
 - `print(myVariable)`
 - `myInFunc()`
 - `myFunc()`

Ficheiros

- Tratamento de ficheiros é importantíssimo em qualquer tipo de aplicação
- Python contém várias funções para criar, ler, atualizar e apagar ficheiros
- Para abrir um ficheiro, usa-se o comando
 - `open(nome_ficheiro, modo)`
- `nome_ficheiro`
 - É a designação do ficheiro juntamente com a sua extensão(ex: teste.txt).
 - Por defeito, a sua localização é a mesma onde está o ficheiro Python do programa que estamos a executar
 - Se o ficheiro se encontrar noutra localização, terá que se colocar o caminho completo do mesmo.
 - Ver Conceitos básicos de Python, como especificar o caminho dos ficheiros

Ficheiros

- modo
 - Existem vários modos como abrir um ficheiro
 - “r” – *Read* (leitura) – Valor por defeito. Abre um ficheiro para leitura. Retorna um erro se não existir
 - “a” – *Append* (acrescentar) – Abre um ficheiro para acrescentar. Se o ficheiro não existir, cria um novo
 - “w” – *Write* (escreve) – Abre um ficheiro para escrita. Se o ficheiro não existir, cria um novo
 - “x” – *Create* (criar) – Cria um ficheiro específico. Retorna um erro se já existe
 - É possível também especificar como o ficheiro é manuseado
 - “t” – *Text* (texto) – Valor por defeito. Modo de texto
 - “b” – *Binary* (binário) – Modo binário. (ex. Imagens)

Ficheiros

- Tipos de ficheiros mais utilizados para leitura/escrita
 - Ficheiro texto .txt: Documento standard de texto que contem apenas texto. Usado para introduzir dados do utilizador ou pode ser utilizado para inicializar programas
 - Ficheiro separado por virgulas .csv: Documento de dados que contém texto, separado por virgulas (Comma Seperated Values). Usado para troca de dados entre diferentes aplicações
 - Ficheiro de base de dados .db: Base de dados de vários tipos, com várias tabelas, podendo existir relações entre elas. Usa-se a linguagem SQL (Structured Query Language) para ler ou escrever neste tipo de ficheiros
 - Ficheiro notações JSON: Contém estruturas simples de dados em formato Javascript Object Notation. Usado para troca de dados entre aplicações, principalmente entre aplicações e servidores web.

