EAD

EDUCAÇÃO A DISTÂNCIA

UNIDADE III

Introdução à Programação Estruturada

Prof. Me. Ricardo Veras

Definindo as Funções

FUNÇÕES:

- Uma função é um bloco de código (um conjunto de códigos) que, somente, vai ser executado se for explicitamente “chamado” no programa;

- Podemos passar os dados ou as informações para as funções (conhecidas como ***parâmetros*** ou ***argumentos***);

- Uma função pode (ou não) ***retornar*** uma informação (um valor) como resultado para quem a chamou (uma variável, por exemplo).

Sintaxe:

- ***def*** **<nome_da_funcao> (<parametros>):**

  <bloco_de_instrucoes>

Definindo as Funções

FUNÇÕES:

- Procura-se definir as funções sempre no início dos programas (convenção);
- Para se dar **nome** a uma função, deve-se seguir as mesmas regras de nomes de variáveis;
- Os **parâmetros** são os valores que a função recebe para processar e são definidos dentro do parêntese. A função pode ter de 0 (zero) a N (diversos) parâmetros;
    - Para se ativar (executar) uma função basta “chamar” pelo seu nome, definindo os valores de todos os seus parâmetros;
    - Os parênteses devem ser colocados, mesmo que a função não receba valor algum como parâmetro.

Definindo as Funções

FUNÇÕES:

- O bloco de instruções define o processamento da função, de modo que ela pode retornar ou não algum valor ao programa (a quem a chamou);

- Este “retorno de valor” é a devolução de uma resposta do processamento da função em forma de valor (caracterizado pela linha de comando com a palavra “***return***”), que pode ser armazenado em uma variável utilizada no programa.

Exemplo de uma função:

```
1   # Função sem retorno
2   def imprime_na_tela(valor):
3       print("Valor enviado:", valor)
4
5   # Corpo do Programa
6   imprime_na_tela("Texto de aviso!...") # ..imprimindo um texto
7   a = 10
8   imprime_na_tela(a) #..imprimindo o valor de uma variável
```

Fonte: autoria própria.

- Esta função **não** retorna valor, pois o seu objetivo é, apenas, a impressão de um valor na tela.

Definindo as Funções

FUNÇÕES:

Outro exemplo:

... esta função retorna à soma dos quadrados de dois valores:

= $a^2 + b^2$

```
1   # Função com retorno
2   def somar_quadrados(x1, x2):
3       quad1 = x1 * x1
4       quad2 = x2 * x2
5       vFinal = quad1 + quad2
6       return vFinal
7   # Início do Programa
8   v1 = float(input("Digite o 1o número: "))
9   v2 = float(input("Digite o 2o número: "))
10  # Pode-se simplesmente imprimir o resultado...:
11  print(somar_quadrados(v1, v2))
12  # ...como também guardá-lo para futuro processamento:
13  result = somar_quadrados(v1, v2)
14  # e com este valor (result) continuar o processamento
```

Fonte: autoria própria.

Definindo as Funções

FUNÇÃO DENTRO DE FUNÇÃO:

- A linguagem Python permite a criação de uma função dentro de outra função (mas que pode ser chamada, apenas, pela primeira função e não pelo corpo principal do programa).

Exemplo:

```
1   def primeira_funcao():
2       def segunda_funcao():
3           print(x + 1)
4       x = 10
5       print(x * a)
6       segunda_funcao()
7   # Início do Programa
8   a = 5
9   # a "chamada" abaixo irá imprimir:
10  #    50
11  #    11
12  primeira_funcao()
13  # a "chamada" comentada abaixo é proibida:
14  # segunda_funcao() - ERRO - não pode
```

Fonte: autoria própria.

Definindo as variáveis locais e globais

VARIÁVEIS LOCAIS E GLOBAIS:

- Uma variável somente estará disponível para o acesso dentro da região (do bloco) onde ela foi criada (definida). Este conceito é chamado de “**escopo**”;

- De acordo com o conceito de escopo, uma variável criada dentro de uma função pertence ao escopo local daquela função, e, portanto, é considerada uma **variável local**, somente podendo ser utilizada dentro daquela mesma função.

```
1  def funcao_com_variavel(par1):
2      x = 20
3      return (x * par1)
4  # Início do Programa
5  valor01 = funcao_com_variavel(5)
6  print(valor01)
7  print(x) # se eu tentar imprimir
8  #          o valor de x, resultará
9  #          em ERRO de programa
```

Fonte: autoria própria.

- Obs.: neste programa, a variável *x* só pode ser utilizada dentro da função.

Definindo as variáveis locais e globais

VARIÁVEIS LOCAIS E GLOBAIS:

- Uma variável criada dentro do corpo principal do programa Python é chamada de **variável global** e pertence ao “escopo global”;

- Variáveis globais estão disponíveis para **qualquer parte do programa Python**, ou seja, dentro de qualquer escopo criado no programa, seja das funções, seja do corpo principal do programa.

Um exemplo:

```
1   def funcao_imprimir_a():
2       print(a)
3   # Início do Programa
4   a = 10
5   # ..primeira impressão (na função)
6   funcao_imprimir_a() # irá imprimir o valor 10
7   # ..segunda impressão (no programa)
8   print(a) # também irá imprimir o valor 10
```

Fonte: autoria própria.

Definindo as variáveis locais e globais

VARIÁVEIS LOCAIS E GLOBAIS:

Outro exemplo:

- ... neste caso, a variável cujo nome é “a” está definida das duas formas: local e global;

```
1   def imprimir_a():
2       a = 12 # esta é uma variável local
3       print(a)
4   # Início do Programa
5   a = 10 # esta é uma variável global
6   # ..primeira impressão
7   imprimir_a() # irá imprimir 12
8   # ..segunda impressão
9   print(a)  # irá imprimir 10
```

Fonte: autoria própria.

- Para a função, como a variável foi definida de forma local, a sua utilização será independente da variável global do corpo principal do programa.

Definindo as variáveis locais e globais

VARIÁVEIS LOCAIS E GLOBAIS:

Mais outro exemplo:

- Neste exemplo, a variável “a” está sendo utilizada de forma global na função. Para isso, explicitamos, na função, que aquela variável “a” utilizada é a variável que existe no corpo principal do programa;

```
1   def imprimir_a():
2       global a
3       a = 12
4       print(a)
5   # Início do Programa
6   a = 10
7   # ..primeira impressão
8   imprimir_a() # irá imprimir 12
9   # ..segunda impressão
10  print(a)  # irá imprimir 12
```

Fonte: autoria própria.

- Agora, na função, como ela foi definida de forma global (palavra reservada global), tudo o que fizermos com a variável poderá repercutir na variável global do corpo principal do programa;
- Obs.: as definições de variáveis locais e globais servem, também, para as funções dentro de funções.

Argumento *default*

ARGUMENTO *DEFAULT*:

- Uma outra forma de enviar os argumentos para as funções, é através da sintaxe argumento = valor. Diferente da forma tradicional de chamar as funções (em que os parâmetros devem seguir a ordem correta), na forma de argumento = valor, a ordem não importará.

Exemplo:

```
1   def dados_usuario(nome, idade, cidade):
2       print("Nome:", nome)
3       print("Idade:", idade)
4       print("Cidade:", cidade)
5   # Chamada "tradicional" (como já vimos)
6   dados_usuario("Maria Aparecida", 25, "Jundiaí")
7   ''' IMPRIMIRÁ:
8       Nome: Maria Aparecida
9       Idade: 25
10      Cidade: Jundiaí
11  '''
12  # Chamada com argumentos definidos (fora de ordem)
13  dados_usuario(idade = 34, cidade = "São Paulo", nome = "João Silva")
14  ''' IMPRIMIRÁ
15      Nome: João Silva
16      Idade: 34
17      Cidade: São Paulo
18  '''
```

Fonte: autoria própria.

Argumento *default*

ARGUMENTO *DEFAULT*:

- É possível utilizar na declaração da função um valor de argumento padrão, caso a função seja chamada **sem o argumento**, e, nesse caso, a função utilizará o valor *default* (padrão) definido.

Exemplo:

```
1   def fruta(fruta= "maça"):
2       print("Gosto de", fruta)
3   fruta("banana")
4   fruta("uva")
5   fruta()
6   ''' IMPRIMIRÁ:
7       Gosto de banana
8       Gosto de uva
9       Gosto de maça
10  '''
```

Fonte: autoria própria.

- Obs.: percebam que, neste caso, o argumento *default* está definido no parâmetro da função.

Interatividade

Um programador precisou criar uma função de nome “fcn_01”, de modo que a mesma devesse receber 3 valores numéricos (a serem recebidos nas variáveis “a”, “b” e “c”, nesta ordem), de modo que o valor “*default*” dessas variáveis sejam, respectivamente, 15, 20 e 30. Qual deve ser a declaração (definição) da função, e para que serve o valor “*default*”?

a) Declaração: def fcn_01 (a = 15, b = 20, c = 30):
   o “valor *default*” define o valor das variáveis caso a chamada da função não os contenha.
b) Declaração: fcn_01 (15, 20, 30):
   o “valor *default*” faz com que as “variáveis” se tornem “constantes”.
c) Declaração: def fcn_01 (a, b, c):
   o “valor *default*” é o valor enviado na chamada da função.
d) Declaração: function fcn_01 (a = 15, b = 20, c = 30):
   o “valor *default*” faz com que as “variáveis” se tornem “constantes”.
e) Declaração: fcn_01 (15, 20, 30):
   o “valor *default*” define o valor das variáveis caso a chamada da função não os contenha.

Resposta

Um programador precisou criar uma função de nome “fcn_01”, de modo que a mesma devesse receber 3 valores numéricos (a serem recebidos nas variáveis “a”, “b” e “c”, nesta ordem), de modo que o valor “*default*” dessas variáveis sejam, respectivamente, 15, 20 e 30. Qual deve ser a declaração (definição) da função, e para que serve o valor “*default*”?

a) Declaração: def fcn_01 (a = 15, b = 20, c = 30):
   o “valor *default*” define o valor das variáveis caso a chamada da função não os contenha.
b) Declaração: fcn_01 (15, 20, 30):
   o “valor *default*” faz com que as “variáveis” se tornem “constantes”.
c) Declaração: def fcn_01 (a, b, c):
   o “valor *default*” é o valor enviado na chamada da função.
d) Declaração: function fcn_01 (a = 15, b = 20, c = 30):
   o “valor *default*” faz com que as “variáveis” se tornem “constantes”.
e) Declaração: fcn_01 (15, 20, 30):
   o “valor *default*” define o valor das variáveis caso a chamada da função não os contenha.

Manipulação de *Strings*

***STRINGS*:**

- Em Python, as *Strings* (do tipo *str*) representam caracteres, palavras e textos, de forma que, na sua declaração, (no código do programa) os seus valores podem estar contidos entre aspas simples ou duplas;

```
1   # Trabalhando com Strings
2   # ..............................
3   txt1 = "Um texto qualquer..."
4   print(txt1)
5   # ..............................
6   txt2 = 'Outro texto qualquer...'
7   print(txt2)
```

Fontes: autoria própria.

- Além dessa forma, podemos representar os textos multilinhas usando, na atribuição, três aspas duplas ou três aspas simples.

```
1   # Trabalhando com Strings
2   # ..............................
3   txtMulti1 = """Este texto pode conter quebra de linhas.
4   Todas as quebras serão guardadas.
5   Imprime-se esta variável como qualquer outra variável."""
6   print(txtMulti1)
7   # ..............................
8   txtMulti2 = '''Este outro texto também pode conter quebra de linhas.
9   Todas as quebras serão guardadas.
10  Imprime-se esta variável como qualquer outra variável.'''
11  print(txtMulti2)
```

Manipulação de *Strings*

***STRINGS*:**

- Podemos concatenar (unir) *strings* umas com as outras, simplesmente, realizando uma "adição" com elas.

Exemplo:

```
1   # Trabalhando com Strings
2   # ..............................
3   a = "Você tem que ser o espelho da mudança que está propondo. "
4   b = "Se eu quero mudar o mundo, "
5   c = "tenho que começar por mim."
6   d = " (Mahatma Gandhi)"
7   txtFinal = a + b + c + d
8   print(txtFinal)
```

... e o resultado (txtFinal) seria uma frase com todos os textos juntos:

```
Você tem que ser o espelho da mudança que está propondo. Se
eu quero mudar o mundo, tenho que começar por mim. (Mahatma
Gandhi)
```

Fontes: autoria própria.

Manipulação de *Strings*

***STRINGS*:**

Podemos, inclusive, realizar a concatenação (união) de *strings* utilizando a forma recursiva:

```
1   # Trabalhando com Strings
2   # ..............................
3   txt = "Você tem que ser o espelho da mudança que está propondo. "
4   txt += "Se eu quero mudar o mundo, "
5   txt += "tenho que começar por mim."
6   txt += " (Mahatma Gandhi)"
7   print(txt)
```

Fonte: autoria própria.

- ... sendo o mesmo resultado do exemplo anterior.

Manipulação de *Strings*

***STRINGS*:**

- Assim como em outras linguagens de programação, as *strings*, em Python, são encaradas como uma **matriz unidimensional de caracteres** (um “vetor” de caracteres);
- Desta forma, os colchetes podem ser usados para acessar os caracteres individuais da *string*, de forma que, dentro dos colchetes, indicamos a posição do caractere desejado (o qual chamaremos de índice);
- Obs.: importante: o primeiro índice da matriz SEMPRE será o 0 (zero).

Ex.:

```
1   # Trabalhando com Strings
2   txt = "humano"
3   print(txt[0])
4   print(txt[1])
5   print(txt[2])
6   print(txt[3])
7   print(txt[4])
8   print(txt[5])
9   print("FIM DO PROGRAMA")
```

Saída:

```
h
u
m
a
n
o
FIM DO PROGRAMA
```

Fontes: autoria própria.

Manipulação de *Strings*

***STRINGS*:**

- Uma função bastante utilizada na manipulação de *strings* é a função **len(...)**. Ela recebe como parâmetro a *string* e retorna o tamanho daquela *string* (quantidade de caracteres que ela possui);

```
1   # Trabalhando com Strings
2   txt = "O gato de botas."
3   tam = len(txt)
4   print(tam) # imprimirá 16
```

Fonte: autoria própria.

- Percebam que tanto os espaços quanto as pontuações são considerados no tamanho da *string*, já que a função **len(...)** retorna a quantidade total de caracteres.

Manipulação de *Strings*

***STRINGS*:**

- É possível usar a instrução ***in*** para verificar se um caractere, uma palavra ou uma frase existem (ou fazem parte) na *string*;
- Quando utilizada, retorna um valor lógico, verdadeiro (*True*) ou falso (*False*) dependendo se aquele pedaço de frase está contido na frase toda;
- Obs.: a instrução ***not in*** é o oposto da instrução ***in***.

```
1   # Trabalhando com Strings
2   txt = "O gato de botas."
3   gOk = "gato" in txt
4   cOk = "cachorro" in txt
5   print(gOk) # imprimirá True
6   print(cOk) # imprimirá False
```

Fonte: autoria própria.

Manipulação de *Strings*

***STRINGS*:**

Observação: o “pedaço” da frase procurado tem que ser exato para ser encontrado:

```
1   # Trabalhando com Strings
2   txt = "O gato de botas."
3   gOk = "Gato" in txt
4   print(gOk) # imprimirá False
```

É muito comum utilizar estes recursos num contexto de condicional, de forma a realizar as ações condicionadas à existência ou não de determinados caracteres na *string*, como, por exemplo:

```
1   # Trabalhando com Strings
2   frase = "A rua está calma hoje!"
3   if ("agitada" not in frase):
4       print("A frase não contem a palavra: agitada")
5   print("FIM DO PROGRAMA")
```

Fontes: autoria própria.

Manipulação de *Strings*

***STRINGS*:**

- O *Fatiamento de Strings* é um poderoso recurso do Python (muito utilizado em DATA SCIENCE, por exemplo);
- O fatiamento funciona com a utilização de “dois-pontos” (:) no índice da *string*.

Sintaxe:

variavel_string [num_esq : num_dir]

- O número à esquerda dos dois pontos indica o índice de início da fatia;
- O número à direita dos dois pontos indica um número depois do índice final da fatia.

Ex.:

```
1   # Trabalhando com Strings
2   frase = "A rua está calma hoje!"
3   print(frase[2:14])
4   # na posição 2 temos o "r" de "rua"
5   # na posição 14 temos o "m" de "calma"
6   # no entanto imprime: "rua está cal"
```

Fonte: autoria própria.

Manipulação de *Strings*

***STRINGS*:**

VARIAÇÕES DO FATIAMENTO DE *STRING*:

- Omitir o número da esquerda representa pegar “do início até o índice determinado (uma posição antes)”;
- Omitir o número da direita representa pegar “exatamente do índice determinado até o final da *string*”;
- Omitir os números (esquerda e direita) irá fazer uma cópia de **todos** os caracteres da *string*;
- Utilizar um valor negativo, indica as posições “a partir da direita” (-1 é o último caractere, -2 é o penúltimo caractere etc.).

```
1   # Trabalhando com Strings
2   frase = "Estratégia"
3   print(frase[:5]) # imprime "Estra"
4   print(frase[6:]) # imprime "égia"
5   print(frase[:]) # imprime "Estratégia"
6   print(frase[-3:]) # imprime "gia"
```

Fonte: autoria própria.

Manipulação de *Strings*

***STRINGS*:**

- As *strings* têm vários métodos prontos para serem usados. Esses métodos funcionam como funções que retornam um valor, mas que não modificam, diretamente, o conteúdo da variável.

A lista desses métodos é extensa, mas alguns deles são os mais utilizados:

```
1   # Trabalhando com Strings
2   frase = "  Mais um dia de SOL  "
3   print(frase.upper())
4   # imprime "  MAIS UM DIA DE SOL  "
5   print(frase.lower())
6   # imprime "  mais um dia de sol  "
7   print(frase.strip())
8   # imprime "Mais um dia de SOL"
9   print(frase.split(" "))
10  # imprime "['', '', 'Mais', 'um', 'dia', 'de', 'SOL', '', '']"
11  print(frase.replace("SOL", "CHUVA"))
12  # imprime "  Mais um dia de CHUVA  "
13  print(frase)
14  # imprime "  Mais um dia de SOL  "
```

Fonte: autoria própria.

Interatividade

Analisando o código a seguir, o que será impresso na tela do console?

```
x = “PaLaVrA”
for n in range(len(x)):
    print(x[(len(x)-1)-n].lower(), end=“”)
# dica:
# … o parâmetro end=“” na função print faz com que, a cada;
# … iteração, o valor seja impresso na mesma linha, e não;
# … em linhas diferentes.
```

a) PaLaVrA.
b) palavra.
c) ARVALAP.
d) ArVaLaP.
e) arvalap.

Resposta

Analisando o código a seguir, o que será impresso na tela do console?

```
x = “PaLaVrA”
for n in range(len(x)):
    print(x[(len(x)-1)-n].lower(), end=“”)
# dica:
# … o parâmetro end=“” na função print faz com que, a cada;
# … iteração, o valor seja impresso na mesma linha, e não;
# … em linhas diferentes.
```

a) PaLaVrA.
b) palavra.
c) ARVALAP.
d) ArVaLaP.
e) arvalap.

Abertura, leitura e gravação em arquivos

ABERTURA DE ARQUIVOS:

Abre-se um arquivo com a função *open*(...), que retorna um objeto do tipo "*file*" (arquivo):

variavel = *open* ("localizacao_do_ arquivo", "modo")

Existem quatro "modos" de operação para se abrir um arquivo:

- "w" (Escrita): abre o arquivo para a escrita e **se o arquivo não existir, ele será criado**;
- "r" (Leitura): valor padrão. Abre o arquivo para a leitura e, neste caso, **se o arquivo não existir mostrará um erro**;
- "a" (Adiciona): abre o arquivo para acrescentar os dados e **se o arquivo não existir, ele será criado**;
- "x" (Cria): cria o arquivo e **se ele já existir mostrará um erro**.

  Para garantirmos a correta conversão dos caracteres do arquivo para o Python (e vice-versa), podemos utilizar o parâmetro ***encoding = "utf-8"*** no método *open*:

  variavel = *open* ("arquivo", "modo", encoding = "utf-8")

Abertura, leitura e gravação em arquivos

ESCREVENDO EM ARQUIVOS:

- A função “*write*( )” é utilizada para escrever ou gravar dados no arquivo. Esta função apaga todo o conteúdo já existente no arquivo, substituindo-o por outro texto.

LENDO ARQUIVOS:

- O método “*read*()” é utilizado para ler um arquivo;
- Pode-se utilizar o método arquivo.readlines(), que irá transformar o arquivo em uma lista dentro do programa (facilitando a sua manipulação).

FECHANDO ARQUIVOS:

- O método “*close*()” serve para fechar o arquivo. O fechamento do arquivo garante a liberação do mesmo, preservando a integridade dos seus dados.

Abertura, leitura e gravação em arquivos

EXEMPLOS DE LEITURA DE ARQUIVO:

Ex. 1:

```
1   # Abrindo o arquivo para leitura
2   arquivo = open("segredo.txt", "r", encoding = "utf-8")
3   # Lendo (e imprimindo) todo o arquivo
4   print(arquivo.read())
5   # Fechando o arquivo ao final do programa
6   arquivo.close()
```

Ex. 2:

```
1   # Abrindo o arquivo para leitura
2   arquivo = open("segredo.txt", "r", encoding = "utf-8")
3   # Lendo apenas os 7 primeiros caracteres do arquivo
4   print(arquivo.read(7))
5   # Fechando o arquivo ao final do programa
6   arquivo.close()
```

Fontes: autoria própria.

Abertura, leitura e gravação em arquivos

EXEMPLOS DE LEITURA DE ARQUIVO:

Ex. 3:

```
1   # Abrindo o arquivo para leitura
2   arquivo = open("segredo.txt", "r", encoding = "utf-8")
3   # Lendo apenas a primeira linha do arquivo
4   print(arquivo.readline())
5   # Lendo agora a segunda e a terceira linha do arquivo
6   print(arquivo.readline())
7   print(arquivo.readline())
8   # Fechando o arquivo ao final do programa
9   arquivo.close()
```

Fonte: autoria própria.

Abertura, leitura e gravação em arquivos

EXEMPLOS DE ESCRITA DE ARQUIVO:

Ex. 1:

```
1   # Abrindo o arquivo para escrita
2   # ..acrescentando um texto
3   arquivo = open("segredos.txt", "a", encoding = "utf-8")
4   arquivo.write("\nA senha é 123456")
5   arquivo.close()
```

Ex. 2:

```
1   # Abrindo o arquivo para escrita
2   # ..criando ou substituindo todo o texto
3   arquivo = open("segredos.txt", "w", encoding = "utf-8")
4   arquivo.write("A senha é 123456")
5   arquivo.close()
```

Fontes: autoria própria.

Manipulação básica de matrizes

MATRIZES:

- Para lidarmos com os cálculos de matrizes em Python, podemos utilizar as listas (que veremos mais à frente). No entanto, algumas funções específicas só podem ser utilizadas se trabalharmos com a biblioteca *NumPy*, que não vem instalada, inicialmente, com o Python.

Para instalar o *NumPy* deve-se:

- Abrir o "*Prompt* de Comando" (no Windows), ou o "Terminal" (no Linux ou MacOs);
- Digitar: ***pip install numpy*;**
  - *NumPy* – É uma biblioteca do Python utilizada, principalmente, para os cálculos numéricos com *Arrays* Multidimensionais.

  Esses cálculos são necessários quando lidamos com:
  - *Machine Learning*;
  - Processamento de Imagem e Computação Gráfica;
  - Tarefas matemáticas (substituindo o MATLAB).

Manipulação básica de matrizes

MATRIZES:

- Sempre que formos utilizar o *NumPy* em nossos programas Python, devemos importar a sua biblioteca;

```
1   # Manipulando Matrizes com Numpy
2   import numpy
3   matriz = numpy.array([1, 2, 3])
4   print(matriz) # imprime: [1 2 3]
```

- Pode-se criar <u>apelidos</u> para as bibliotecas importadas; assim, não precisamos escrever o nome completo da biblioteca toda vez que formos utilizá-la.

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   matriz = np.array([1, 2, 3])
4   print(matriz) # imprime: [1 2 3]
```

Ex. de Matriz 2D:

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   # Matriz 2D ("lista de lista") - matriz 2x3
4   matriz = np.array([[1, 2, 3], [4, 5, 6]])
5   print(matriz)
6   # imprimirá:
7   # [[1 2 3]
8   #  [4 5 6]]
```

Fontes: autoria própria.

Manipulação básica de matrizes

MATRIZES:

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   m1 = np.array(10) # dimensão zero - um número
4   m2 = np.array([1, 2, 3, 4]) # dimensão um vetor - de tamanho 4
5   m3 = np.array([[1, 2, 3], [4, 5, 6]]) # matriz 2x3
6   m4 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]) # matriz 2x2x3
7   print(m1.ndim) # imprimirá 0
8   print(m2.ndim) # imprimirá 1
9   print(m3.ndim) # imprimirá 2
10  print(m4.ndim) # imprimirá 3
```

Fonte: autoria própria.

- É possível criar matrizes (*arrays*) de N dimensões utilizando as listas dentro de listas e assim por diante;
- Obs.: a propriedade ***ndim*** retorna à “dimensão” (quantidade de dimensões) da matriz.

Manipulação básica de matrizes

MATRIZES:

- Para acessar um elemento da matriz, o fazemos da mesma maneira como vimos para acessar um caractere de uma *string*. A indexação de cada uma das dimensões das matrizes também inicia em zero.

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   m1 = np.array([22, 17, -3, 11, 25, 6])
4   # para acessar o primeiro elemento da matriz
5   print(m1[0]) # imprimirá o n.o 22
6   # para acessar o terceiro elemento da matriz
7   print(m1[2]) # imprimirá o n.o -3
8   # para acessar o último elemento da matriz
9   print(m1[-1]) # imprimirá o n.o 6
10   # para matrizes multidimensionais (ex.: m2 = 2x4)
11   m2 = np.array([[22, 17, -3, 11], [-1, 4, 25, 6]])
12   # para acessar o segundo elemento do primeiro grupo
13   print(m2[0, 1]) # imprimirá o n.o 17
```

Fontes: autoria própria.

Manipulação básica de matrizes

OPERAÇÕES COM MATRIZES:

- As operações de matrizes com o Python seguem as mesmas regras sobre as matrizes matemáticas.

Ex. 1:

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   # matrizes multidimensionais (ex.: m1 = 2x4)
4   m1 = np.array([[22, 17, -3, 11], [-1, 4, 25, 6]])
5   # para multiplicar uma matriz por um escalar
6   m2 = m1 * 5
7   print(m2)
8   # imprimirá:
9   # [[110  85 -15  55]
10  #  [ -5  20 125  30]]
```

Fonte: autoria própria.

Manipulação básica de matrizes

OPERAÇÕES COM MATRIZES:

Ex. 2:

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   # matrizes multidimensionais (ex.: m1 = 2x4)
4   m1 = np.array([[22, 17, -3, 11], [-1, 4, 25, 6]])
5   m2 = np.array([[10, 2, 7, -4], [-1, 12, 20, 3]])
6   # para somar matrizes (mesma ordem)
7   m3 = m1 + m2
8   print(m3)
9   # imprimirá:
10  # [[32 19  4  7]
11  #  [-2 16 45  9]]
```

Fonte: autoria própria.

Manipulação básica de matrizes

OPERAÇÕES COM MATRIZES:

Ex. 3:

```
1   # Manipulando Matrizes com Numpy
2   import numpy as np
3   m1 = np.array([[2, -1, 3], [1, 4, -2]]) # 2x3
4   m2 = np.array([[3, 1], [-3, 2], [5, -2]]) # 3x2
5   # multiplicando matrizes
6   m3 = np.dot(m1, m2)
7   print(m3)
8   # imprimirá: [[ 24  -6]
9   #             [-19  13]]
10  m4 = np.dot(m2, m1)
11  print(m4)
12  # imprimirá: [[  7   1   7]
13  #             [ -4  11 -13]
14  #             [  8 -13  19]]
```

Fonte: autoria própria.

Interatividade

De acordo com o código a seguir, qual é o valor da ordem da matriz “mat” e qual é o texto que será impresso na tela do console (com a função *print*)?

```
import numpy as np
mat = np.array([
    [[2, 14, 8, 1], [17, 4, 13, 9]],
    [[10, 5, 11, 18], [26, 19, 23, 16]],
    [[15, 6, 27, 3], [12, 20, 0, 21]],
])
print(mat[1, 0, 3] * 2)
```

a) Ordem da matriz = 4 x 3 x 2, e será impresso: [2, 0, 6].

b) Ordem da matriz = 2 x 3 x 4, e será impresso: 18.

c) Ordem da matriz = 4 x 3 x 2, e será impresso: 36.

d) Ordem da matriz = 3 x 2 x 4, e será impresso: [2, 0, 6].

e) Ordem da matriz = 3 x 2 x 4, e será impresso: 36.

Resposta

De acordo com o código a seguir, qual é o valor da ordem da matriz “mat” e qual é o texto que será impresso na tela do console (com a função *print*)?

```
import numpy as np
mat = np.array([
    [[2, 14, 8, 1], [17, 4, 13, 9]],
    [[10, 5, 11, 18], [26, 19, 23, 16]],
    [[15, 6, 27, 3], [12, 20, 0, 21]],
])
print(mat[1, 0, 3] * 2)
```

a) Ordem da matriz = 4 x 3 x 2, e será impresso: [2, 0, 6].

b) Ordem da matriz = 2 x 3 x 4, e será impresso: 18.

c) Ordem da matriz = 4 x 3 x 2, e será impresso: 36.

d) Ordem da matriz = 3 x 2 x 4, e será impresso: [2, 0, 6].

e) Ordem da matriz = 3 x 2 x 4, e será impresso: 36.

Estruturas de dados

SEQUÊNCIAS:

- Sequência é um conceito importante em Python, pois as estruturas de dados de sequências possuem funções e métodos específicos para a sua manipulação;
- O Python trabalha com 6 (seis) estruturas de sequências, que são: *strings*, tuplas, objetos de *range* (faixa numérica), listas, *bytes* e matrizes de *bytes*;
- A principal característica das sequências é a sua ordem determinística, que a difere das outras estruturas de dados.

Estruturas de dados

LISTAS:

Lista é uma coleção “ordenada” e “mutável” de elementos, delimitados por colchetes:

- **Ordenada** quer dizer que existe uma sequência e uma ordem (é indexada);
- **Mutável** quer dizer que podemos modificar os valores dos elementos;
- Uma lista permite a inclusão de elementos de diversos tipos;
- Uma lista permite a inclusão de elementos duplicados.

```
1   lista_1 = [ 65, True, 'humano', 3.333, ["outra", 'lista'], True, 65]
2   print(lista_1)
3   print('O tamanho da lista 1 é:', len(lista_1))
4   print('O tipo da lista 1 é:', type(lista_1))
5
6   '''
7   Saída:
8   [65, True, 'humano', 3.333, ['outra', 'lista'], True, 65]
9   O tamanho da lista 1 é: 7
10  O tipo da lista 1 é: <class 'list'>
11  '''
```

Fonte: autoria própria.

Estruturas de dados

TUPLAS:

Tupla é uma coleção também “**ordenada**”, porém “**imutável**” de elementos, delimitados por parênteses:

- **Imutável** quer dizer que não podemos modificar os valores dos seus elementos;
- Geralmente, são utilizadas para guardar itens em uma variável simples;
- Não é possível modificar as tuplas de nenhuma forma (adicionar, remover ou modificar itens);
- Seus itens são acessados da mesma forma que as listas.

```
1   minha_tupla = ("fusca", "monza", "opala", "corsa", "palio")
2   print(minha_tupla)
3   ''' Saída:
4   ('fusca', 'monza', 'opala', 'corsa', 'palio')
5   '''
6   print(type(minha_tupla))
7   ''' Saída: <class 'tuple'>
8   '''
```

Fonte: autoria própria.

Estruturas de dados

TUPLAS:

Exemplo:

```
1   minha_tupla = ("fusca", "monza", "opala", "corsa", "palio")
2   print(minha_tupla)
3   ''' Saída:
4   ('fusca', 'monza', 'opala', 'corsa', 'palio')
5   '''
6   # como não é possível alterar os seu valores
7   # podemos transformar a tupla em lista alterar
8   # o seu valor, e depois retorná-la para tupla
9   minha_lista = list(minha_tupla)
10  print(minha_lista)
11  minha_lista[3] = "passat"
12  print(minha_lista)
13  ''' Saída:
14  ['fusca', 'monza', 'opala', 'passat', 'palio']
15  '''
16  minha_tupla = tuple(minha_lista)
17  print(minha_tupla)
18  ''' Saída:
19  ('fusca', 'monza', 'opala', 'passat', 'palio')
20  '''
21
```

Fonte: autoria própria.

Estruturas de dados

BYTES E *BYTEARRAY* (SEQUÊNCIA BINÁRIA):

- As sequências de *bytes* são objetos imutáveis que guardam 1 *byte* por elemento, ou seja, cada posição pode receber um valor inteiro de 0 até 255. A função *bytes*() pode receber uma lista de inteiros;
- Já os objetos ***bytearray*** são sequências mutáveis;
- São sequências de inteiros no intervalo 0 < = x < 256;
- Objetos ***bytes*** e ***bytearray*** oferecem diversos métodos que são válidos quando trabalhamos com os dados compatíveis com *ASCII*;
    - Muitos protocolos binários importantes são baseados em codificação *ASCII* (de texto);
    - São bastante úteis para a estrutura de dados ou protocolos de comunicação baseados em texto.

Estruturas de dados

BYTES E *BYTEARRAY* (SEQUÊNCIA BINÁRIA):

```
a = bytes([0,1,2,3,11,15, 17, 255])
# os bytes são representados em hexadecimal
print(a)
'''
Saída:
b'\x00\x01\x02\x03\x0b\x0f\xff'
'''
```

```
x = "Esta é uma String (str)"
print("x:", x)
b = bytes(x, "utf-8") # bytes(fonte, codificação)
# Obs.: codificação = codificação binária de
#       representação de caracteres
print("b:", b)
'''
Saída:
x: Esta é uma String (str)
b: b'Esta \xc3\xa9 uma String (str)'
'''
```

Fontes: autoria própria.

```
a = bytearray(8)
print(a)
# modificando o primeiro byte
a[0] = 255
print(a)
'''
Saída:
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
bytearray(b'\xff\x00\x00\x00\x00\x00\x00\x00')

'''
```

Estruturas de dados

CONJUNTO (*SET*):

- Conjuntos, ou *set* (em Python), são estruturas de dados que implementam as operações de Matemática que lidam com os conjuntos: união, intersecção, diferença, entre outras;

- Uma das características dos *sets* é não admitir a repetição de elementos. Em Python, conjuntos são **mutáveis** e **não** são **indexados** (não mantém a ordem de seus elementos – não necessariamente);

    - Um *set* pode ser criado a partir de listas, tuplas e qualquer outra estrutura de dados que seja enumerável;

    - Também pode ser criado diretamente (literalmente) utilizando “chaves”.

Estruturas de dados

CONJUNTO (*SET*):

Operações com conjuntos:

- União – realizada pelo operador “|” (*pipe*): ***cj1 | cj2;***
- Intersecção – realizada pelo operador “&” (“& comercial”): ***cj1 & cj2;***
- Diferença – que utiliza o operador “–” (menos): ***cj1 – cj2.***

- Conjuntos possuem a propriedade tamanho (número de elementos), que pode se obtida com o uso da função “**len()**”.

Estruturas de dados

CONJUNTO (*SET*):

Exemplo:

```
1   lista1 = [2, 5, 4, 7, -2, 5, 7]
2   tupla1 = (6, 8, 1, -2, 1, 7, 0, 9)
3   cj0 = {5, 8, 2} # criação literal de conjunto
4   cj1 = set(lista1) # criação a partir de lista
5   print("cj1:", cj1)
6   cj2 = set(tupla1) # criação a partir de tupla
7   print("cj2:", cj2)
8   cj3 = cj1 | cj2 # união de conjuntos
9   print("cj3:", cj3)
10  cj4 = cj1 & cj2 # intersecção de conjuntos
11  print("cj4:", cj4)
12  cj5 = cj1 - cj0 # subtração de conjuntos
13  print("cj5:", cj5)
14  '''
15  Saídas:
16  cj1: {2, 4, 5, 7, -2}
17  cj2: {0, 1, 6, 7, 8, 9, -2}
18  cj3: {0, 1, 2, 4, 5, 6, 7, 8, 9, -2}
19  cj4: {-2, 7}
20  cj5: {4, -2, 7}
21  '''
```

Fonte: autoria própria.

Interatividade

Analisando as variáveis criadas a seguir (**ed1**, **ed2** e **ed3**), qual é o tipo da estrutura de dados que cada uma delas representa (respectivamente)?

```
ed1 = (3, 7, 1, 0, 2, 8, 5)
ed2 = {3, 7, 1, 0, 2, 8, 5}
ed3 = [3, 7, 1, 0, 2, 8, 5]
```

a) Representam: conjunto (ou *set*), tupla e lista.
b) Representam: tupla, conjunto (ou *set*) e lista.
c) Representam: lista, conjunto (ou *set*) e tupla.
d) Representam: tupla, lista e conjunto (ou *set*).
e) Representam: lista, conjunto (ou *set*) e lista.

Resposta

Analisando as variáveis criadas a seguir (**ed1**, **ed2** e **ed3**), qual é o tipo da estrutura de dados que cada uma delas representa (respectivamente)?

```
ed1 = (3, 7, 1, 0, 2, 8, 5)
ed2 = {3, 7, 1, 0, 2, 8, 5}
ed3 = [3, 7, 1, 0, 2, 8, 5]
```

a) Representam: conjunto (ou *set*), tupla e lista.
b) Representam: tupla, conjunto (ou *set*) e lista.
c) Representam: lista, conjunto (ou *set*) e tupla.
d) Representam: tupla, lista e conjunto (ou *set*).
e) Representam: lista, conjunto (ou *set*) e lista.

Referências

- Imagens das telas do VS *Code*: autoria própria.

ATÉ A PRÓXIMA!