Introdução à Inteligência Artificial

Módulo 01: Introdução à Inteligência Artificial (IA)

Professor: Dr. João Paulo R. R. Leite

UNIFEI Softex MCTI FUTURO
FUTURO DO TRABALHO, TRABALHO DO FUTURO
GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Sumário

Introdução ao Python

UNIFEI Softex MCTI FUTURO

FUTURO DO TRABALHO, TRABALHO DO FUTURO

GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Objetivo principal

- O **objetivo** desta aula é introduzir a linguagem de programação **Python** de maneira bastante **prática**. Serão apresentados alguns conceitos introdutórios sobre a linguagem, assim como sua sintaxe básica, habilitando o aluno a desenvolver programas simples com estrutura **imperativa**.

Introdução

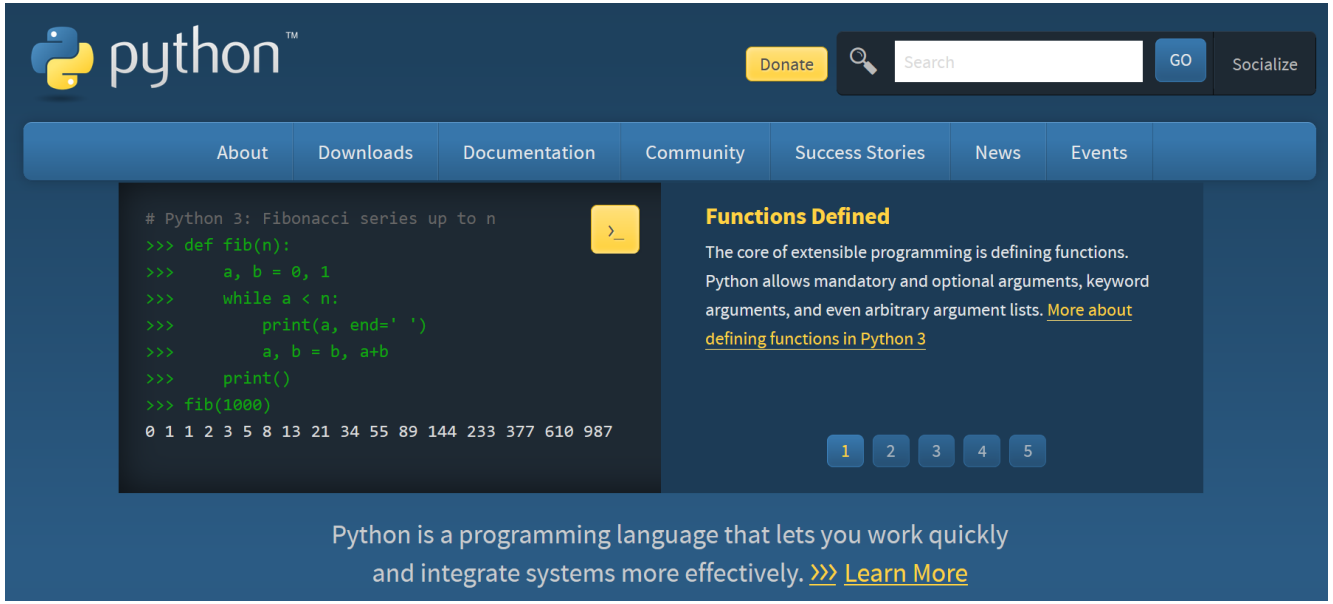
Python é uma linguagem de programação **orientada a objetos** *open-source* de **sintaxe bastante simples**, com suporte a vários paradigmas e estilos de programação (imperativo, funcional).

Foi criada por **Guido Van Rossum** em 1989, tornada pública em 1991 e atualmente está em sua **versão 3** (que utilizaremos neste curso e não é mais compatível com a versão 2, descontinuada em 2020).

A linguagem Python é uma linguagem **interpretada** e de **alto nível**.

Uma **linguagem interpretada** não gera um programa executável. Os códigos são executados linha por linha por um **interpretador**. Isso faz com que seu desempenho seja inferior ao de linguagens compiladas como C e C++. Mais informações e recursos podem ser obtidos em www.python.org/

Linguagens de alto nível são linguagens mais parecidas com linguagens humanas, e são mais **fáceis de aprender e ler**. Seu contraponto são as linguagens de baixo nível, mais próximas da linguagem de máquina.



Aplicações e Bibliotecas

- Inteligência Artificial
  - Machine Learning (scikit-learn, TensorFlow, PyTorch)
- Ciência de Dados
  - NumPy, Pandas, matplotlib
- Desenvolvimento de Software
  - Paradigmas POO e Funcional
- Desenvolvimento Web
  - Django, Tornado, Flask
- *Web Crawler*
  - Scrapy, BeautifulSoup
- E muito mais... Tornando a linguagem ainda mais poderosa.

Let’s **Code!**

Estruturas Básicas da Linguagem

É importante saber dialogar com o usuário da aplicação, através de comandos de entrada e saída. Uma **entrada** é qualquer informação externa fornecida ao programa (através do teclado, p.ex.), enquanto uma **saída** é qualquer informação enviada ao mundo externo pelo programa (na tela do dispositivo, p.ex.).

Entradas: serão lidas do teclado através da função **input()**.
Saídas: serão escritas na tela utilizando a função **print()**.

```
# first python program
name = input("Enter your name: ")
print("Hello,", name, "!")
```

```
Enter your name: John
Hello, John !
```

Do exemplo, podemos ver que:

- **Comentários** podem ser incluídos no código e são iniciados com o caractere **"#"**. Eles devem ser sucintos e significativos, sem exageros.
- Dê nomes significativos às suas variáveis. O programa ganha muito em legibilidade.

Para trabalhar com dados em qualquer linguagem de programação, é necessário **armazená-los na memória** do dispositivo, mesmo que temporariamente. A reserva de memória, no programa, é realizada através da declaração de variáveis.

Para criar uma variável, basta que dar-lhe um nome e um valor inicial, utilizando o operador de atribuição (=). **Não é necessário especificar um tipo**. Ele é **inferido** pela atribuição. Por exemplo:

```
a = 10 # creates variable "a" and assigns an int value (10)
b, c = 20, a # creates variables b and c, with values 20 and 10 (a)
print(a, b, c)
```

```
10 20 10
```

O valor da variável **pode ser modificado** posteriormente em seu programa (inclusive o tipo):

```
a = 1
print(a)
a = 2
print(a)
```

```
1
2
```

```
a = 1
print(a)
a = "Now a String"
print(a)
```

```
1
Now a String
```

Que nomes podemos dar às variáveis?

- Os nomes de variáveis podem conter letras, o caractere "_" e números, mas não podem ser iniciados por um número.
- A linguagem é *case sensitive*, ou seja, diferencia maiúsculas e minúsculas. "Nome" e "nome" são variáveis diferentes.
- Além disso, as palavras reservadas da linguagem não podem ser utilizadas para nomear variáveis. São as seguintes:

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Tipos básicos de dados

- **Inteiro (int)**: quantidades contáveis, sem parte fracionária. Ex.: 1, 5, -25, 1000.
- **Real (float)**: Números que podem conter partes fracionárias. Ex.: 1.0, -2.8, 3.1415, 8.74.
- **Lógico (bool)**: Podem possuir apenas dois valores, verdadeiro ou falso. Pode ter dois valores: True e False.
- **Strings (str):** Expressam textos. Ex.: “Hello World”, “Brasil”, “123456”, “AI”. Em Seus valores são expressos entre aspas duplas, como nos exemplos.
- **Complexos (complex):** São escritos na forma x + yj, em que x é a parte real e y a parte imaginária, representando números complexos. Ex.: 5 + 6j, 1.2 + 5.7j.

Operações básicas

Operações **aritméticas** básicas:

Operador	Descrição	Exemplo
+	Soma	2 + 3 (== 5)
-	Subtração	5 - 2 (== 3)
*	Multiplicação	2 * 5 (== 10)
/	Divisão	15.0 / 2.0 (== 7.5)
%	Resto da divisão inteira	5 % 2 (== 1)
**	Expoente	5**2 (== 25)
//	Divisão inteira	5 // 2 (== 2)

Operações **lógicas** básicas:

Operador	Descrição	Exemplo
==	Igual	a == b
!=	Diferente	a != b
>	Maior que	a > b
<	Menor que	a < b
>=	Maior ou igual a	a >= b
<=	Menor ou igual a	a <= b
and	“e” lógico	a and b
or	“ou” lógico	a or b
not	Negação lógica	not a

Conversões entre tipos de dados podem ser realizadas colocando o nome do tipo desejado e a informação ou a variável a ser convertida entre parênteses. Exemplo:

```
val = "100" # a string
val2 = 100 + val
print("Result = ", val2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-2d4d191aac20> in <module>
      1 val = "100" # a string
----> 2 val2 = 100 + val
      3 print("Result = ", val2)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Não consegue somar valores de tipos **int** e **str**.

```
val = "100" # a string
val2 = 100 + int(val)
print("Result = ", val2)
```

```
Result =  200
```

O que aconteceria se eu convertesse 100 pra **str**?

Essa conversão pode ser muito útil na entrada via teclado.

Todos os valores obtidos pela função **input**() são recebidos como ***strings***. Para realizar operações aritméticas sobre eles, é necessário convertê-los para os tipos adequados. Veja:

```
radius = float(input("Type the circle radius: ")) # gets str input and converts it to float
area = 3.14159265*(radius**2)
print("Circle area =", area)
```

```
Type the circle radius: 10
Circle area = 314.159265
```

Exercício:

Vamos escrever um programa em Python que calcula a média final dos alunos que fizeram 2 provas (70%) e um trabalho (30%). O programa deverá receber as 3 notas do tipo “float” através do teclado e escrever a média na tela. As notas podem ter partes fracionárias.

Em Python, strings são sequências de caracteres, e possuem algumas características particulares.

Escape Sequence

Para inserir caracteres que são ilegais em uma String, utilize um caractere de “escape”, ou seja, a barra invertida \. Por exemplo, se quisermos inserir o caractere aspas duplas “ em uma string, é necessário que seja antecedido por uma \.

```
s = "python""
print(s)
```

```
  File "<ipython-input-16-89604c9d16d2>", line 1
    s = "python""
                ^
SyntaxError: EOL while scanning string literal
```

```
s = "python\""
print(s)
```

```
python"
```

Raw String

Se quisermos ignorar as possíveis “*escape sequences*” dentro de uma string, devemos escrever a letra r antes dela. Isso é útil quando a string possui barras invertidas importantes para seu entendimento, como em caminhos para arquivos:

```
s = r"C:\users\JP\python\project\abc.txt"
print(s)
```

```
C:\users\JP\python\project\abc.txt
```

Operações com Strings:

Concatenação e repetição:

```
s1 = "hello "
s2 = "world"
print(s1 + s2) # + concatenates
```

```
hello world
```

```
s1 = "hello "
print(s1 * 3) # * replicates
```

```
hello hello hello
```

Algumas funções úteis:

```
s = "python is AWESOME!"
print(s.upper()) # to uppercase
print(s.lower()) # to lowercase
```

```
PYTHON IS AWESOME!
python is awesome!
```

```
s.split()
```

```
['python', 'is', 'AWESOME!']
```

```
s.replace("AWESOME", "great")
```

```
'python is great!'
```

```
conective = "--"
phrase = conective.join(["Artificial", "Intelligence"])
print(phrase)
```

```
Artificial--Intelligence
```

```
print(phrase[3]) # prints 4th character in string
```

```
i
```

```
#string formatting
print("My name is %s, age %d" %("John", 37))
```

```
My name is John, age 37
```

Vetores e Matrizes

Para representar conjuntos de variáveis relacionadas através de um nome único e índices, o Python utiliza as **Listas**. Veja:

```
a = [] # creates empty list
b = [1,3,7,8,9,6,5,4,2,0]  # creates list with pre-defined values
c = [0 for i in range(100)] # create 100-element list, filled with zeros
```

Acesso a elementos:

```
print(b[7]) # gets 8th element (index starts with 0)
print(b[0:5]) # gets elements from 0 to 5
print(b[0:5:2]) # gets elements from 0 to 5 with step = 2
```

```
4
[1, 3, 7, 8, 9]
[1, 7, 9]
```

```
print(b[:3]) # list slice (up to 3)
print(b[3:]) # list slice (from 3 on)
print(b[::2]) #list slice (from begin to end, step 2)
```

```
[1, 3, 7]
[8, 9, 6, 5, 4, 2, 0]
[1, 7, 9, 5, 2]
```

Concatenação e repetição:

```
a = [1,2,3]
b = [4,5]
```

```
c = a + b # concat
print(c)
c = a * 3 #repeat
print(c)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

As listas apresentam também alguns **métodos** que podem ser muito úteis:

```
animals = ["cat", "horse", "dog"]
animals.append("bird") # inserts at the end
print(animals)
```

```
['cat', 'horse', 'dog', 'bird']
```

```
animals.remove("cat") # removes item from list
print(animals)
```

```
['horse', 'dog', 'bird']
```

```
animals.insert(2, "fish") # inserts at index 2
print(animals)
```

```
['horse', 'dog', 'fish', 'bird']
```

```
animals.sort() #sorts the list
print(animals)
```

```
['bird', 'dog', 'fish', 'horse']
```

```
animals.reverse()
print(animals)
```

```
['horse', 'fish', 'dog', 'bird']
```

Mas e as **matrizes**?
Em Python, elas são representadas como listas de listas. Veja:

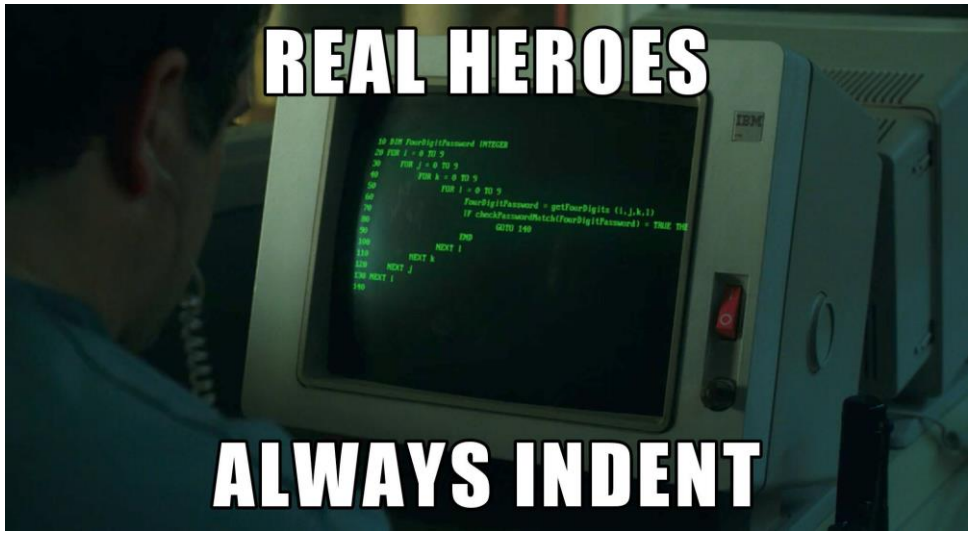
```
#representing a matrix
# [[0,1]
#  [2,3]
#  [4,5]]
mat = [[0,1], [2,3], [4,5]]
print(mat) # whole matrix
print(mat[2][1]) # getting specific element
mat[1][0] = 100 # changing element value
print(mat)
```

```
[[0, 1], [2, 3], [4, 5]]
5
[[0, 1], [100, 3], [4, 5]]
```

Indentação

Um dos pilares da programação imperativa é a **execução sequencial** das instruções, com a possiblidade de desvios de fluxo baseados em condições e/ou repetições de **blocos de comandos**.

Blocos de comando são grupos de instruções que devem ser tratados como uma unidade. Em outras linguagens, como C/C++, são identificados e agrupados por chaves (“{ }”). Em Python, esses blocos são agrupados através apenas da **indentação**, ou seja, espaços inseridos com a tecla *tab* no início de cada linha. Ou seja, instruções em um “**mesmo nível de indentação**” pertencem a um mesmo bloco de comandos.

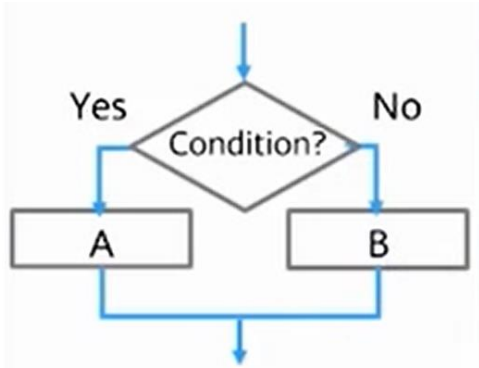


Em Python, estruturas de decisão do tipo “se” serão tratadas com o comando **if** ou **if-else:**

```
if(condicao):
    comando1
    comando2
    ...
    comandon
```

```
if (condicao):
    comando1
    comando2
else:
    comando3
    comando4
```

Duas coisas chamam atenção:
Dois pontos (:)
Indentação
Fique atento!



Exemplo:

```
p1 = float(input("Entre com nota 1: "))
p2 = float(input("Entre com nota 2: "))
media =  (p1+p2) / 2

if(media >= 6):
    print("Aprovado! Media = ", media)
else:
    print("Reprovado! Media = ", media)
```

```
Entre com a nota 1: 7
Entre com a nota 2: 8
Aprovado! Media =  7.5
```

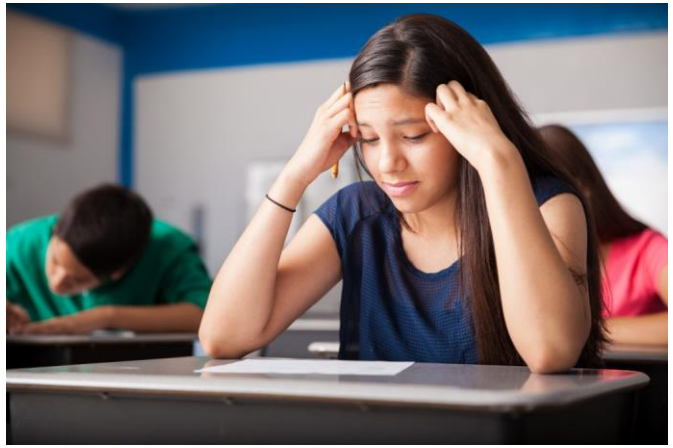
É possível “**aninhar**” estruturas condicionais, bastando para isso manter a consistência da indentação. Imagine que queiramos acrescentar uma nota de recuperação para o aluno reprovado:

```
p1 = float(input("Entre com nota 1: "))
p2 = float(input("Entre com nota 2: "))
media =  (p1 + p2) / 2

# estruturas if-else aninhadas
if (media >= 6):
    print("Aprovado! Media = ", media)
else:
    rec = float(input("Entre com nota da recuperacao: "))
    mediarec = (media + rec) / 2
    if (mediarec >= 6):
        print("Aprovado! Media = ", mediarec)
    else:
        print("Reprovado! Media = ", mediarec)

#instrução fora dos blocos if-else. Nível de indentação.
print("Fim do Programa!")
```

```
Entre com a nota 1: 5
Entre com a nota 2: 6
Entre com a nota da recuperacao: 7
Aprovado! Media =  6.25
```



Há uma novidade com relação a outras linguagens: quando vamos testar vários valores, com várias condições diferentes (faixas, etc.), podemos utilizar o comando **elif**, que é uma combinação de **else + if**, deixando o **código mais compacto** (evita aninhamento de muitos *ifs*).

```
if (condicao1):
    comando1
elif (condicao2):
    comando2
    comando3
elif (condicao3):
    comando4
else:
    comando5
    comando6
```

```
if(condicao1):
    comando1
else:
    if (condicao2):
        comando2
        comando3
    else:
        if (condicao3):
            comando4
        else:
            comando5
            comando6
```

```
ang = float(input("Entre com o angulo: "))

if (ang < 0 or ang > 360):   # verificacao
    print("Angulo invalido")
elif (ang >0 and ang <= 90):
    print("Quadrante 1")
elif (ang <= 180):
    print("Quadrante 2")
elif (ang <= 270):
    print("Quadrante 3")
else:
    print("Quadrante 4")
```

Em certas situações, desejamos que trechos de código sejam repetidos até que uma condição seja satisfeita. Pra isso, utilizamos as **estruturas de repetição**, ou ***loops***.

Somente como lembrete, é fundamental que o critério de parada de uma estrutura de repetição seja bem definido. Um *loop infinito* fará com que o código não progrida e o programa trave (ou esgote os recursos da máquina).

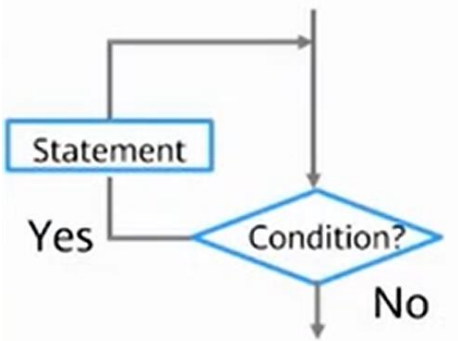
Em Python, existem dois tipos de ***loops***:

- Quando há uma quantidade indefinida de repetições;
  - **while**
- Quando essa quantidade é conhecida.
  - ***for***

Para o primeiro tipo, são utilizadas estruturas do tipo “**enquanto**”, através do comando **while** do Python:

```
# não se esqueça dos dois pontos!
while (condicao):
    comando1 # o bloco a repetir é definido pela indentação
    comando2
    ...
    comandon
```

Se necessário, também podem estar aninhadas.



Exemplo:

```
soma = nota = cont = 0
while (nota >= 0):   # repete enquanto não digitar valor negativo
    nota = float(input("Entre com a nota: "))
    if (nota >= 0):
        soma = soma + nota
        cont += 1
media = soma/cont  # ultima não conta
print("Media da turma = ", media)
```

```
Entre com a nota: 10
Entre com a nota: 8
Entre com a nota: 9
Entre com a nota: -1
Media da turma = 9.0
```

Para o segundo tipo, quando se conhece a quantidade de **iterações** (repetições) a serem realizadas, é utilizada uma variável como contador. Essas estruturas com contadores necessitam de três parâmetros:

- Valor inicial do contador, recebido antes da primeira execução;
- Valor final, que, ao ser atingido pelo contador, fará com que a repetição seja interrompida;
- Incremento, que é o valor somado ao contador a cada repetição.

Essas estruturas são do tipo **para**:

para (contador) de (valor inicial) até (valor final) com passo (incremento) faça:

A estrutura utilizada em Python para este tipo de tarefa é o **for**, e utiliza a função **range** para trabalhar com o contador.

A função range pode receber 1, 2 ou 3 parâmetros:

range(y)
Nesse caso, y será considerado como **valor final** (exclusivo). O valor inicial será considerado igual a 0 e o passo será considerado como 1. Ou seja, a função gerará valores inteiros entre 0 e y-1.

range(x, y)
Aqui, x será considerado o **valor inicial** (inclusivo) e y será o **valor final** (exclusivo). O passo continua a ser igual a 1. Irá gerar números inteiros entre x e y-1.

range(x, y, z)
Utilizado quando há necessidade de se **especificar um passo diferente** de 1, através do valor z. Funciona da mesma maneira que o anterior, mas gera números na sequência x, (x + z), (x + 2z)...

Um exemplo:

```
print("Utilizando for-range:")
for i in range(0,10,2): # para i de 0 a 10, com passo 2, faça:
    print(i)

print("O mesmo com while:")
i = 0   # inicializacao do contador
while(i < 10):  # condicao
    print(i)
    i += 2      # incremento
```

```
Utilizando for-range:
0
2
4
6
8
O mesmo com while:
0
2
4
6
8
```

A partir dessa estrutura, podemos percorrer uma matriz de forma automatizada, como o fizemos com vetores, a partir de **dois loops aninhados**: Um para percorrer as "sublistas" (primeiro índice) e outro para seus elementos (segundo índice).

```
mat = [[3,1], [4,1], [5,9], [2,6]]
for i in range(4):
    for j in range(2):
        print(" ", mat[i][j], end='') # ao invés de pular linha, finaliza com ''
    print() # para pular a linha
```

```
 3  1
 4  1
 5  9
 2  6
```

Caso não seja necessário trabalhar com os índices das listas, é possível percorrê-las de uma outra maneira, bastante intuitiva. Veja os exemplos:

```
lista = [0, 2, 4, 6, 8]
for elemento in lista:    # para cada "elemento" em "lista"
    print(" ", elemento, end='')
```

```
matriz = [[0,1,2], [3,4,5], [6,7,8]]
for listan in matriz:   # para cada lista na matriz
    for elemento in listan:   # para cada elemento dessa lista
        print(" ", elemento, end='')
    print()
```

Funciona como uma estrutura "*for each*", presente em outras linguagens.

Exercício:
Escreva um programa em Python que calcule a soma de todos os elementos de uma progressão aritmética (P.A.), que inicia em 2 e não tem valores maiores que 100, com razão igual a 3.

Exercício:
Escreva um programa em Python que leia uma matriz de inteiros de 3x3 e calcule a soma da sua diagonal principal. Utilize as estruturas de repetição adequadas.

Exercício:
Escreva um programa em Python em que o professor digita as notas de uma turma de alunos e as guarda em uma lista. Ao final, é calculada a média da turma, que é escrita na tela. No início, o professor entra com a quantidade de alunos.

Apoio

Este projeto é apoiado pelo Ministério da Ciência, Tecnologia e Inovações, com recursos da Lei nº 8.248, de 23 de outubro de 1991, no âmbito do [PPI-Softex| PNM-Design], coordenado pela Softex.

João Paulo Reus Rodrigues Leite
Universidade Federal de Itajubá
e-mail: joaopaulo@unifei.edu.br

UNIFEI

Softex

MCTI FUTURO
FUTURO DO TRABALHO, TRABALHO DO FUTURO

GOVERNO FEDERAL
BRASIL
UNIÃO E RECONSTRUÇÃO