



PYTHON – AULA 4

LÓGICA DE PROGRAMAÇÃO APLICADA



AGENDA

Aula 1

Instalando e conhecendo o Python e as ferramentas do curso.
Revisitando lógica de programação em Python.

Aula 2

Revisitando lógica de programação em Python –
continuação.

Aula 3

Trabalhando com listas, tuplas e dicionários.
Leitura e escrita de JSON.

Aula 4

Leitura e escrita de arquivos.
Introdução a Numpy e Pandas.

AGENDA

Aula 5

Orientação a objeto no Python.

Aula 6

Trabalhando com os algoritmos.
Ordenação e Recursão. – Pesquisa em largura.

Aula 7

Algoritmo de Dijkstra.


Aula 8

Algoritmos Genéticos.



PROGRAMANDO EM PYTHON

LENDO
ARQUIVOS DE TEXTO



Ao longo dos desafios que você vai encontrar no mundo digital, certamente vai precisar utilizar algum tipo de fonte de dados.

E os arquivos de texto – ainda que obedecendo a alguma formatação específica – são fontes de dados bem comuns!

Vamos aprender hoje como escrever algo dentro de um arquivo de texto.

Para isso, crie um novo script chamado `lendo_texto.py`



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")  
print(arquivo.read())
```

A função *open* é uma função nativa do Python. O objetivo dessa função é retornar para dentro do nosso script um objeto do tipo *arquivo* (que estamos guardando em um objeto chamado *arquivo*).

Porém, ao executar o script acima, o Python deve nos avisar de um erro: o arquivo NÃO EXISTE!

Por isso, crie dentro do seu computador um arquivo de texto chamado “arquivo_de_texto.txt” dentro do diretório c:\arquivos.



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")  
  
print(arquivo.readline())  
  
print(arquivo.readline())
```

Cada vez que o método *readline* for executado, uma nova linha do nosso arquivo será lida.

Essa abordagem pode ser útil para quando quisermos ler a primeira linha, por exemplo. Mas e se quisermos ler várias linhas desse arquivo?

Podemos criar um loop!



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")
```

```
for linha in arquivo.readlines():  
    print(linha)
```

Com o loop acima somos capazes de passar por cada uma das linhas do nosso arquivo de texto e imprimir essas linhas na tela. Isso ocorre porque o método *readlines()* retorna um objeto do tipo *list* e nosso loop *for* faz a iteração por todas as linhas dessa lista.

Para irmos mais além, podemos colocar cada uma das linhas do nosso arquivo em uma lista! Assim podemos manipular sua ordem como quisermos!



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")

linhas_do_arquivo = arquivo.readlines()

print("Ei! Eu consegui transformar meu arquivo em uma {}".format(type(linhas_do_arquivo)))

linhas_do_arquivo.sort()

print(linhas_do_arquivo)
```

Uma vez que o método *readlines()* retorna uma list, podemos armazenar o seu retorno dentro de uma lista e depois manipulá-la, inclusive colocando as “linhas” (entre aspas, pois são itens de uma lista) em ordem!

Apesar de estarmos lendo o conteúdo do arquivo de texto com muita facilidade, estamos cometendo um erro muito grande: é preciso FECHAR o arquivo após utilizar.

Isso pode ser resolvido com o método *close()* do nosso objeto chamado *arquivo*.



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")

linhas_do_arquivo = arquivo.readlines()

print("Ei! Eu consegui transformar meu arquivo em uma {}".format(type(linhas_do_arquivo)))

linhas_do_arquivo.sort()

print(linhas_do_arquivo)

arquivo.close()
```

Após fazermos o uso de um recurso, é importante encerrarmos a conexão com ele para que fique liberado para outros processos.


No nosso caso, quem faz a função de encerrar o uso do arquivo de texto que abrimos é o método `.close()`.



PROGRAMANDO EM PYTHON



ESCREVENDO EM
ARQUIVOS DE TEXTO



Agora que já aprendemos a utilizar a função *open* para ler o conteúdo dos arquivos de texto, vamos usá-la para gravar conteúdos nesses arquivos.

Quando usamos a função *open*, além do caminho do arquivo que queremos abrir podemos indicar mais um argumento: o modo de abertura desse arquivo.

Vamos entender quais são os modos disponíveis:

Abaixo temos uma lista com todos os modos de abertura que podemos passar como argumento na função *open*

VALOR	SIGNIFICADO
r	abrir para leitura (modo padrão)
w	abrir para a escrita, sobrescrevendo o conteúdo
x	abrir para a criação de arquivo, gerando uma falha se existir um arquivo de mesmo nome
a	abrindo para escrita, anexando o novo conteúdo ao final do conteúdo já existente no arquivo
b	abrir em modo binário
t	abrir em modo de texto (modo padrão)
+	abrir para atualização (escrita e leitura)

Que tal escrevermos um script chamado *escrevendo_texto.py* dentro do nosso projeto?



```
conteudo = "Estou testando criar um arquivo de texto. Então, estou... textando?"
```

```
arquivo = open("c:\\arquivos\\novo_arquivo.txt", "w")
```

```
arquivo.write(conteudo)
```

```
arquivo.close()
```

Armazenamos o conteúdo que desejamos gravar dentro do arquivo de texto em uma variável do tipo String.



```
conteudo = "Estou testando criar um arquivo de texto. Então, estou...  
textando?"
```

```
arquivo = open("c:\\arquivos\\novo_arquivo.txt", "a")
```

```
arquivo.write(conteudo)
```

```
arquivo.close()
```

Se mudarmos o modo de abertura do arquivo de *w* para *a*, a cada execução do script o conteúdo será acrescentado no arquivo de texto.



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")

for linha in arquivo.readlines():
    print(linha)
```

Com o loop acima somos capazes de passar por cada uma das linhas do nosso arquivo de texto e imprimir essas linhas na tela. Isso ocorre porque o método *readlines()* retorna um objeto do tipo *list* e nosso loop *for* faz a iteração por todas as linhas dessa lista.

Para irmos mais além, podemos colocar cada uma das linhas do nosso arquivo em uma lista! Assim podemos manipular sua ordem como quisermos!



```
arquivo = open("c:\\arquivos\\arquivo_de_texto.txt")

linhas_do_arquivo = arquivo.readlines()

print("Ei! Eu consegui transformar meu arquivo em uma {}".format(type(linhas_do_arquivo)))

linhas_do_arquivo.sort()

print(linhas_do_arquivo)

arquivo.close()
```

Após fazermos o uso de um recurso, é importante encerrarmos a conexão com ele para que fique liberado para outros processos.

No nosso caso, quem faz a função de encerrar o uso do arquivo de texto que abrimos é o método `.close()`.

Abaixo temos uma lista com todos os modos de abertura que podemos passar como argumento na função *open*

VALOR	SIGNIFICADO
r	abrir para leitura (modo padrão)
w	abrir para a escrita, sobrescrevendo o conteúdo
x	abrir para a criação de arquivo, gerando uma falha se existir um arquivo de mesmo nome
a	abrindo para escrita, anexando o novo conteúdo ao final do conteúdo já existente no arquivo
b	abrir em modo binário
t	abrir em modo de texto (modo padrão)
+	abrir para atualização (escrita e leitura)



```
conteudo = "Estou testando criar um arquivo de texto. Então, estou... textando?"  
  
arquivo = open("c:\\arquivos\\novo_arquivo.txt", "w")  
  
arquivo.write(conteudo)  
  
arquivo.close()
```

O objeto chamado *arquivo*, por ter recebido o resultado da função *open*, agora conta com o método *write* que recebe como argumento um conteúdo (armazenado na variável *conteudo*) para gravar dentro do arquivo de texto.



```
conteudo = "Estou testando criar um arquivo de texto. Então, estou... textando?"  
  
arquivo = open("c:\\arquivos\\novo_arquivo.txt", "a")  
  
arquivo.write(conteudo)  
  
arquivo.close()
```


Se mudarmos o modo de abertura do arquivo de *w* para *a*, a cada execução do script o conteúdo será acrescentado no arquivo de texto.



PROGRAMANDO EM PYTHON



UM PEQUENO
EXTRA



Por mais simples que seja utilizarmos a função *open* para abrir arquivos de texto, o risco de não fecharmos esses arquivos é grande.

Por isso você pode utilizar o comando *with*, que garante que um recurso que foi aberto seja finalizado.

Veja como é simples usar esse comando para ler um arquivo de texto



#o with usará o open para abrir o arquivo indicado, dentro do objeto arquivo e fará sozinho o encerramento do acesso quando a ultima linha de código dentro dele for executada

```
with open("c:\\arquivos\\arquivo_de_texto.txt", "r") as arquivo:
```

#aqui devemos escrever todos os códigos que usam o arquivo aberto, pois após a última linha de código dentro dessa estrutura, o arquivo será automaticamente encerrado

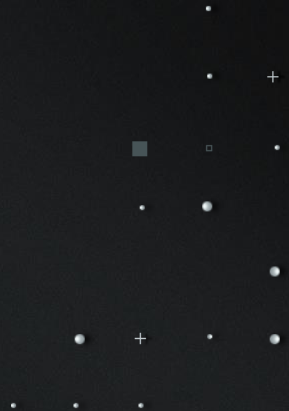
```
    print(arquivo.read())
```

Com apenas duas linhas somos capazes de abrir o arquivo de texto, exibir seu conteúdo e fechar o arquivo.

Isso ocorre porque o comando `with` utiliza a função `open` para abrir o arquivo dentro do objeto `arquivo` e se responsabiliza por fechar esse arquivo ao final. A mesma estrutura pode ser utilizada para gravar um arquivo de texto, pois o `with` também se responsabilizará por encerrar o arquivo quando deixar de ser utilizado.



O PACOTE NUMPY






PROGRAMANDO EM PYTHON



INSTALANDO O
NUMPY.



Uma das grandes forças do Python está na vasta gama de pacotes adicionais que podem ser utilizados para desenvolver seus scripts.

Um dos mais famosos é o NumPy. O NumPy trata-se de uma biblioteca voltada computação científica.

A base do NumPy é a estrutura chamada *ndarray*, capaz de armazenar valores e ser utilizada para diversas operações que vão de ordenação até operações de álgebra linear.

Vamos começar instalando o NumPy.

Esse procedimento pode ser feito de duas formas básicas: instalando a biblioteca no seu computador OU instalando em um ambiente virtual dentro de um projeto feito em Python.

Ao instalar o NumPy no seu computador, você poderá utilizá-lo em todos os projetos que escrever.

Ao instalar em um ambiente virtual dentro de um projeto específico, apenas aquele projeto acessará a biblioteca.

INSTALANDO O NUMPY NO COMPUTADOR

```
Prompt de Comando
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Emerson Abraham>pip --version
pip 22.2.1 from C:\Users\Emerson Abraham\AppData\Local\Programs\Python\Python310\lib\site-packages\pip (python 3.10)

C:\Users\Emerson Abraham>
```

- Abra um Terminal (cmd, no Windows) e verifique se você tem instalado gerenciador de pacotes chamado *pip*. Para isso, basta escrever *pip --version* e teclar Enter.
- Caso seja exibida a versão do pip, você pode seguir para o próximo passo.
- Caso seja exibido um erro, você pode baixar o seguinte script (<https://bootstrap.pypa.io/get-pip.py>) e executar em seu terminal o comando `python get-pip.py`, na pasta onde o arquivo foi salvo.
- Por padrão o pip é instalado a partir do python 3.4

INSTALANDO O NUMPY NO COMPUTADOR

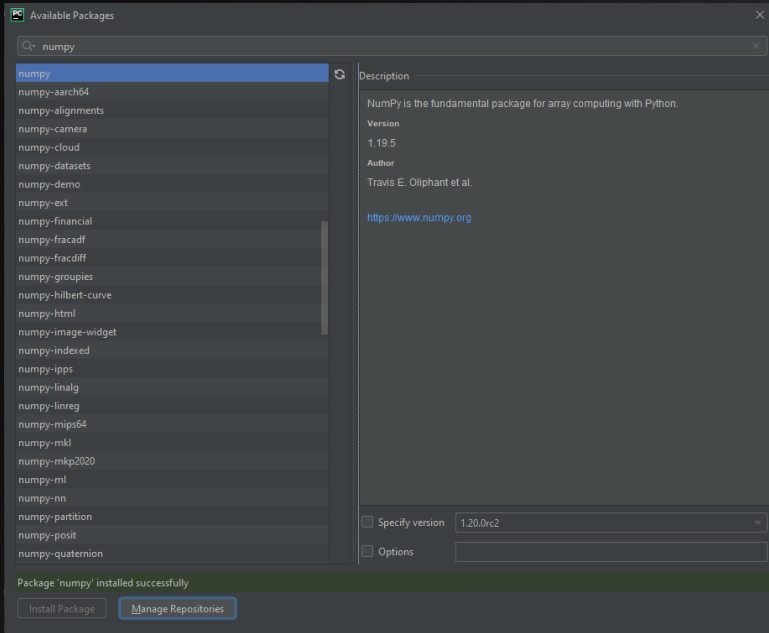
```
CA Prompt de Comando
Microsoft Windows [versão 10.0.19044.1889]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Emerson Abraham>pip install numpy
Collecting numpy
  Using cached numpy-1.23.2-cp310-cp310-win_amd64.whl (14.6 MB)
Installing collected packages: numpy
Successfully installed numpy-1.23.2

C:\Users\Emerson Abraham>
```

- Uma vez que o pip estiver instalado, você poderá obter as bibliotecas através dele.
- No caso do numpy, a instalação deve ser feita através da linha: `pip install numpy`

INSTALANDO O NUMPY EM UM PROJETO



- Vamos considerar a existência de um projeto já chamado conhecendoNumPy aberto dentro do PyCharm
- Acesse o menu File → Settings. Depois selecione o nome do seu projeto do lado esquerdo e “Python Interpreter”.
- Selecione o botão de +
- Na caixa de busca, procure o pacote desejado (numpy, no nosso caso) e clique em install package.



PROGRAMANDO EM PYTHON



CONHECENDO O
NDARRAY



A estrutura de dados básica do *NumPy* é um array multidimensional e homogêneo, ou seja, é uma estrutura capaz de armazenar múltiplos valores de um mesmo tipo.

Como o Python também possui um tipo chamado *array*, mas que é unidimensional, vamos nos referir ao *array* do NumPy como *ndarray*.

Crie um script chamado `ndarray01.py` para começarmos a entender melhor essa biblioteca!



```
import numpy as np
```

```
novo_array = np.array([1,2,3])
```

```
print(novo_array)
```

Para utilizarmos os recursos do NumPy, precisamos importa-lo. Uma forma inteligente de fazer a importação é dando um “apelido” ao pacote. Dessa forma, poderemos escrever o apelido ao invés do nome original.

O apelido que demos é “np”.

Uma vez que temos um ndarray criado, podemos recuperar algumas informações importantes sobre eles.

VALOR	SIGNIFICADO
ndim	Retorna a quantidade de dimensões (eixos) do array
shape	Retorna as dimensões do array. O resultado é uma tupla contendo o tamanho de cada dimensão.
size	Retorna a quantidade de elementos presentes no array.
dtype	Retorna o tipo dos elementos presentes dentro do array.
itemsize	Retorna o tamanho em bytes dos elementos do array.

Vamos testar todas essas informações!



```
import numpy as np

novo_array = np.array([1,2,3])
print("Abaixo está seu array")
print(novo_array)

print("Esse é um ndarray com {} dimensões".format(novo_array.ndim))
print("As dimensões desse ndarray possuem os seguintes tamanhos: {}".format(novo_array.shape))
print("Ao todo, esse ndarray contém {} elementos".format(novo_array.size))
print("Os elementos presentes nesse ndarray são do tipo {}".format(novo_array.dtype))
print("Os elementos presentes nesse ndarray ocupam {} bytes".format(novo_array.itemsize))
```

Para visualizarmos o resultado de cada uma das propriedades, vamos exibir em diferentes prints o valor que retornam



```
import numpy as np
```

```
novο_array = np.array([1,2,3])  
print("Abaixo está seu array")  
print(novo_array)
```

```
print("Esse é um ndarray com {} dimensões".format(novo_array.ndim))  
print("As dimensões desse ndarray possuem os seguintes tamanhos: {}".format(novo_array.shape))  
print("Ao todo, esse ndarray contém {} elementos".format(novo_array.size))  
print("Os elementos presentes nesse ndarray são do tipo {}".format(novo_array.dtype))  
print("Os elementos presentes nesse ndarray ocupam {} bytes".format(novo_array.itemsize))
```

Essa mensagem indicará que nosso array tem apenas 1 dimensão.



```
import numpy as np

novo_array = np.array([1,2,3])
print("Abaixo está seu array")
print(novo_array)

print("Esse é um ndarray com {} dimensões".format(novo_array.ndim))
print("As dimensões desse ndarray possuem os seguintes tamanhos: {}".format(novo_array.shape))
print("Ao todo, esse ndarray contém {} elementos".format(novo_array.size))
print("Os elementos presentes nesse ndarray são do tipo {}".format(novo_array.dtype))
print("Os elementos presentes nesse ndarray ocupam {} bytes".format(novo_array.itemsize))
```

A tupla que representa os tamanhos das nossas dimensões é a (3,). Ou seja, esse array tem apenas 1 dimensão que contém 3 elementos.



```
import numpy as np
```

```
novο_array = np.array([1,2,3])  
print("Abaixo está seu array")  
print(novo_array)
```

```
print("Esse é um ndarray com {} dimensões".format(novo_array.ndim))  
print("As dimensões desse ndarray possuem os seguintes tamanhos: {}".format(novo_array.shape))  
print("Ao todo, esse ndarray contém {} elementos".format(novo_array.size))  
print("Os elementos presentes nesse ndarray são do tipo {}".format(novo_array.dtype))  
print("Os elementos presentes nesse ndarray ocupam {} bytes".format(novo_array.itemsize))
```

Essa mensagem indicará que nosso array contém 3 elementos ao todo.



```
import numpy as np

novo_array = np.array([1,2,3])
print("Abaixo está seu array")
print(novo_array)

print("Esse é um ndarray com {} dimensões".format(novo_array.ndim))
print("As dimensões desse ndarray possuem os seguintes tamanhos: {}".format(novo_array.shape))
print("Ao todo, esse ndarray contém {} elementos".format(novo_array.size))
print("Os elementos presentes nesse ndarray são do tipo {}".format(novo_array.dtype))
print("Os elementos presentes nesse ndarray ocupam {} bytes".format(novo_array.itemsize))
```

Os elementos do nosso array são números inteiros, do tipo int32.



```
import numpy as np
```

```
novο_array = np.array([1,2,3])  
print("Abaixo está seu array")  
print(novo_array)
```

```
print("Esse é um ndarray com {} dimensões".format(novo_array.ndim))  
print("As dimensões desse ndarray possuem os seguintes tamanhos: {}".format(novo_array.shape))  
print("Ao todo, esse ndarray contém {} elementos".format(novo_array.size))  
print("Os elementos presentes nesse ndarray são do tipo {}".format(novo_array.dtype))  
print("Os elementos presentes nesse ndarray ocupam {} bytes".format(novo_array.itemsize))
```

Cada elemento desse array ocupa um espaço de 4 bytes.

A função *array* que nós usamos a pouco para criar nosso primeiro ndarray aceita receber uma lista ou uma tupla do python para converter em ndarray.

Vamos fazer a experiência de criar arrays de diferentes formas utilizando essa função.

Crie o script ndarray02.py para fazermos isso.



```
import numpy as np

array_tupla = np.array([(1,2,3), (4,5,6)])
print("Abaixo está seu array formado através de tuplas")
print(array_tupla)

array_lista = np.array([[1,2,3], [4,5,6]])
print("Abaixo está seu array formado através de listas")
print(array_lista)
```

Para formar um novo array partindo das nossas duas tuplas – (1,2,3) e (4,5,6) – nós passamos uma lista com as duas tuplas como parâmetro para a função.



```
import numpy as np

array_tupla = np.array([(1,2,3), (4,5,6)])
print("Abaixo está seu array formado através de tuplas")
print(array_tupla)

array_lista = np.array([[1,2,3], [4,5,6]])
print("Abaixo está seu array formado através de listas")
print(array_lista)
```

Para formar um novo array partindo das nossas duas listas – [1,2,3] e [4,5,6] – nós passamos uma lista com as duas listas como parâmetro para a função.



```
import numpy as np
```

```
array_tupla = np.array([(1,2,3), (4,5,6)])
print("Abaixo está seu array formado através de tuplas")
print(array_tupla)
```

```
array_lista = np.array([[1,2,3], [4,5,6]])
print("Abaixo está seu array formado através de listas")
print(array_lista)
```

Em ambos os casos, o resultado é a criação de um ***ndarray*** que contém 2 dimensões, uma com dois elementos (2 *linhas*) e outra com 3 elementos (3 *colunas*).

Ao total são 6 elementos do tipo inteiro, cada um ocupando 4bytes.

Suponha agora que você queira GERAR um novo ndarray.

Podemos fazer isso preenchendo o ndarray com zeros, com uns, com valores aleatórios ou criando um arranjo de números pré-definidos.

Crie o script ndarray03.py para fazermos isso.



```
import numpy as np

array_zerado = np.zeros((3, 4))
print("O array gerado com zeros ficou assim:")
print(array_zerado)

array_de_uns = np.ones((5,2), dtype=np.int32)
print("O array gerado com uns ficou assim:")
print(array_de_uns)

array_aleatorio = np.empty((6,3), dtype=np.int32)
print("O array gerado com valores aleatórios ficou assim:")
print(array_aleatorio)

array_gerado = np.arange(10)
print("O array gerado com  ficou assim:")
print(array_gerado)
```

Para gerar um novo array com zeros, podemos utilizar a função `zeros`, que recebe como argumento as dimensões do novo array.

No nosso caso, geramos um novo ndarray com 3 “linhas” e 4 “colunas”, totalmente preenchido com zeros.



```
• • • • • +
• • •import numpy as np
• +
+ • array_zerado = np.zeros((3, 4))
  • print("O array gerado com zeros ficou assim:")
  • print(array_zerado)
  •
  array_de_uns = np.ones((5,2), dtype=np.int32)
  print("O array gerado com uns ficou assim:")
  print(array_de_uns)
|
+ array_aleatorio = np.empty((6,3), dtype=np.int32)
  print("O array gerado com valores aleatórios ficou assim:")
  print(array_aleatorio)

array_gerado = np.arange(10)
print("O array gerado com  ficou assim:")
print(array_gerado)
```

Para gerar um novo array com uns, podemos utilizar a função *ones*, que recebe como argumento as dimensões do novo array.

No nosso caso, geramos um novo ndarray com 5 “linhas” e 2 “colunas”, totalmente preenchido com uns.

Note que utilizamos o parâmetro *dtype* para definir o tipo dos dados inseridos no array.



```
import numpy as np

array_zerado = np.zeros((3, 4))
print("O array gerado com zeros ficou assim:")
print(array_zerado)

array_de_uns = np.ones((5,2), dtype=np.int32)
print("O array gerado com uns ficou assim:")
print(array_de_uns)

array_aleatorio = np.empty((6,3), dtype=np.int32)
print("O array gerado com valores aleatórios ficou assim:")
print(array_aleatorio)

array_gerado = np.arange(10)
print("O array gerado com ficou assim:")
print(array_gerado)
```

Para gerar um novo array com números aleatórios, podemos utilizar a função *empty*, que recebe como argumento as dimensões do novo array.

No nosso caso, geramos um novo ndarray com 6 “linhas” e 3 “colunas”, totalmente preenchido com valores aleatórios.

O tipo indicado foi, novamente, o *int32*, portanto os números aleatórios serão inteiros.



```
• • • • • +
• • •import numpy as np
• + •array_zerado = np.zeros((3, 4))
+ • print("O array gerado com zeros ficou assim:")
  print(array_zerado)
  •
array_de_uns = np.ones((5,2), dtype=np.int32)
print("O array gerado com uns ficou assim:")
print(array_de_uns)
|
+ array_aleatorio = np.empty((6,3), dtype=np.int32)
  print("O array gerado com valores aleatórios ficou assim:")
  print(array_aleatorio)

array_gerado = np.arange(10)
print("O array gerado com  ficou assim:")
print(array_gerado)
```

A função `arange`, por sua vez, parece muito com a função *range* que já utilizamos anteriormente.

Ela aceita um valor inicial, um valor final e um valor de incremento. No nosso caso, estamos gerando valores entre 0 e 9.

Mas o array gerado possui apenas uma dimensão. Para mudarmos isso, precisaremos de uma nova função.

A função *reshape* é capaz de modificar as dimensões de um ndarray.

Ela aceita como parâmetro a nova divisão de dimensões.

Portanto a linha *nome_do_array.reshape(x,y,z)* vai redimensionar o array apontado.

Vamos encadear a chamada dessa função à função *arange* que usamos anteriormente:



```
• • • • • +
• • •import numpy as np
• + •array_zerado = np.zeros((3, 4))
+ • print("O array gerado com zeros ficou assim:")
  print(array_zerado)
  •
array_de_uns = np.ones((5,2), dtype=np.int32)
print("O array gerado com uns ficou assim:")
print(array_de_uns)
|
+ array_aleatorio = np.empty((6,3), dtype=np.int32)
  print("O array gerado com valores aleatórios ficou assim:")
  print(array_aleatorio)

array_gerado = np.arange(10).reshape(2,5)
print("O array gerado com  ficou assim:")
print(array_gerado)
```

Com a chamada da função *reshape*, nosso array original de 1 dimensão com valores entre 0 e 9, agora possui duas dimensões: 2 linhas e 5 colunas.



PROGRAMANDO EM PYTHON

OPERAÇÕES COM
NDARRAYS



Agora que já sabemos criar nossos próprios arrays ou gera-los através das funções que a biblioteca nos dá, vamos a prender a realizar operações com arrays e entre eles.

Para testarmos as potencialidades do NumPy, crie um arquivo chamado `ndarray04.py`



```
• • • • • +
• • •import numpy as np
• +
• array_inicial = np.array([[499.50, 25.00, 35.48], [350.15,30.12,17.3]])
+ • array_resultado = array_inicial - 2
  print("O ndarray resultado após a operação de subtração é")
  print(array_resultado)

array_resultado = array_inicial * 2
print("O ndarray resultado após a operação de multiplicação é")
print(array_resultado)

array_resultado = array_inicial ** 2
print("O ndarray resultado após a operação de exponenciação é")
print(array_resultado)

array_resultado = array_inicial > 40
print("O ndarray resultado após a operação de comparação é")
print(array_resultado)
```

Ao realizar uma operação entre o ndarray e um número, a operação é realizada com cada um dos elementos do ndarray



```
• • • • • +
• • • import numpy as np
• + • array_a = np.array([[0,1], [2,0]])
+ • array_b = np.array([[2,0], [4,3]])
array_resultado = array_a - array_b
print("O ndarray resultado após a operação de subtração é")
print(array_resultado)

array_resultado = array_a * array_b
print("O ndarray resultado após a operação de multiplicação dos elementos")
print(array_resultado)

array_resultado = array_a @ array_b
print("O ndarray resultado após a operação de multiplicação entre duas matrizes é")
print(array_resultado)
```

As operações também podem ser realizadas entre dois ndarrays



```
array_a = np.array([[0,1], [2,0]])

print("A soma dos elementos do ndarray é")
print(array_a.sum())

print("O menor elemento do ndarray é")
print(array_a.min())

print("O maior elemento do ndarray é")
print(array_a.max())

print("A soma de cada coluna dos elementos do ndarray é")
print(array_a.sum(axis=0))
```

Além dessas funções, você pode testar as seguintes:

all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, invert, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where


O PACOTE PANDAS



PROGRAMANDO EM PYTHON



INSTALANDO O
PANDAS.



O processo de instalação do pandas é o mesmo do numpy:

Ou você utiliza o comando *pip install pandas* para instalar no seu computador


Ou você utiliza o menu File → Settings → Nome do Projeto → Python Interpreter → + → Nome do pacote (pandas, nesse caso) → Install package.



PROGRAMANDO EM PYTHON



CONHECENDO AS ESTRUTURAS DO
PANDAS.



O pacote pandas conta com 3 estruturas principais: Séries; Dataframes; e Painéis.

As **Séries** (series) são arrays unidimensionais, tal qual uma tabela de uma única coluna.

Os **dataframes** são como arrays de duas dimensões, similar a uma planilha.

Os painéis (**panel objects**) são dicionários de dataframes, similares a um arquivo do excel.



```
import pandas as pd
import numpy as np

serie = pd.Series([1, 3, 5, 6, 9])

print("Essa é uma estrutura de série no pandas")
print(serie)

indice = np.array(['A', 'B', 'C', 'D', 'E'])

serie = pd.Series([1, 3, 5, 6, 9], index=indice)

print("Essa é uma estrutura de série com um índice nomeado no pandas")
print(serie)
```

A criação de uma nova série no pandas se dá através da instrução “Series”



```
import pandas as pd
import numpy as np

serie = pd.Series([1, 3, 5, 6, 9])

print("Essa é uma estrutura de série no pandas")
print(serie)

indice = np.array(['A', 'B', 'C', 'D', 'E'])

serie = pd.Series([1, 3, 5, 6, 9], index=indice)

print("Essa é uma estrutura de série com um índice nomeado no pandas")
print(serie)
```

Também é possível criar uma série com um índice nomeado



```
import pandas as pd
import numpy as np

indice_datas = pd.date_range("20200101", periods=6)

print("Abaixo está a estrutura que contém o índice das datas")
print(indice_datas)

array_aleatorio = np.empty((6,4), dtype=np.int32)
print("Abaixo está nosso array aleatorio que representa os dados do dataframe")
print(array_aleatorio)

dataframe = pd.DataFrame(array_aleatorio, index=indice_datas, columns=['A', 'B', 'C', 'D'])
print("Abaixo está nosso dataframe")
print(dataframe)
```

É possível utilizar o método `date_range` para criar um intervalo de datas. Usaremos esse intervalo como índice do nosso dataframe.



```
import pandas as pd
import numpy as np

indice_datas = pd.date_range("20200101", periods=6)

print("Abaixo está a estrutura que contém o índice das datas")
print(indice_datas)

array_aleatorio = np.empty((6,4), dtype=np.int32)
print("Abaixo está nosso array aleatorio que representa os dados do dataframe")
print(array_aleatorio)

dataframe = pd.DataFrame(array_aleatorio, index=indice_datas, columns=['A', 'B', 'C', 'D'])
print("Abaixo está nosso dataframe")
print(dataframe)
```

O método *empty* do numpy está sendo usado para criar os dados que simularemos no nosso dataframe



```
import pandas as pd
import numpy as np

indice_datas = pd.date_range("20200101", periods=6)

print("Abaixo está a estrutura que contém o índice das datas")
print(indice_datas)

array_aleatorio = np.empty((6,4), dtype=np.int32)
print("Abaixo está nosso array aleatorio que representa os dados do dataframe")
print(array_aleatorio)

dataframe = pd.DataFrame(array_aleatorio, index=indice_datas, columns=['A', 'B', 'C', 'D'])
print("Abaixo está nosso dataframe")
print(dataframe)
```

O dataframe é composto dos dados do nosso array, tendo como índice das linhas as datas que foram geradas e as colunas as letras A, B, C e D.



O dataframe pode comportar dados de tipos diferentes.

Por essa característica, uma das formas mais interessantes de compormos essa estrutura é através de um dicionário!





```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está nosso dataframe construído a partir de um dicionário")
print(dataframe)
```


Como conteúdo do dicionário que forma nosso dataframe colocamos as colunas como chaves e os dados como valores.



PROGRAMANDO EM PYTHON



VISUALIZANDO UM
DATAFRAME.



- Os dataframes são muito poderosos e possuem uma série de funções para a visualização de dados.

Crie um arquivo chamado `data_frame_view.py` para explorarmos essas funções



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O método *head()* exibe os elementos da parte de cima do dataframe



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O método *head()* exibe os elementos da parte de baixo do dataframe



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O atributo *index* exibe os índices da parte do dataframe



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O atributo *columns* exibe as colunas da parte do dataframe



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O método `to_numpy()` retorna os elementos do dataframe como um ndarray



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O método *describe()* realiza operações de estatística básica



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo está a parte de cima do nosso dataframe")
print(dataframe.head())

print("Abaixo está a parte de baixo do nosso dataframe")
print(dataframe.tail(2))

print("Abaixo estão os índices do nosso dataframe")
print(dataframe.index)

print("Abaixo estão as colunas do nosso dataframe")
print(dataframe.columns)

print("Abaixo está nosso dataframe convertido para um array numpy")
print(dataframe.to_numpy())

print("Abaixo estão estatísticas básicas sobre nosso dataframe")
print(dataframe.describe())

print("Abaixo está nosso dataframe ordenado por uma coluna")
print(dataframe.sort_values(by="D"))
```

O método `sort_values()` ordena os dados de uma coluna



PROGRAMANDO EM PYTHON



OPERAÇÕES INTERESSANTES COM
DATAFRAME.



- Existem diversas operações que podemos realizar com dataframes.

Crie um arquivo chamado `exibições_data_frame.py` para explorarmos algumas dessas operações.



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)

print("Abaixo exibimos os dados da coluna C")
print(dataframe["C"])

print("Abaixo exibimos apenas um intervalo de linhas")
print(dataframe[0:2])

print("Abaixo exibimos apenas os valores do dataframe em que a coluna D seja maior do que 7")
print(dataframe[dataframe["D"] > 7])

print("Abaixo exibimos apenas os valores em que a coluna E contém o valor 'usado'")
print(dataframe[dataframe["E"].isin(["usado"])])
```

Podemos exibir os dados de uma coluna específica, indicando entre colchetes o valor dessa coluna.



```
import pandas as pd +
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Abaixo exibimos os dados da coluna C")
print(dataframe["C"])

print("Abaixo exibimos apenas um intervalo de linhas")
print(dataframe[0:2])

print("Abaixo exibimos apenas os valores do dataframe em que a coluna D seja maior do que 7")
print(dataframe[dataframe["D"] > 7])

print("Abaixo exibimos apenas os valores em que a coluna E contém o valor 'usado'")
print(dataframe[dataframe["E"].isin(["usado"])])
```

Um intervalo de linhas pode ser especificado, indicando primeiro a partir de que linha o corte deve ser feito seguido da quantidade de linhas. Nesse caso serão exibidas as duas primeiras.



```
• • import pandas as pd +
• • import numpy as np
• + dataframe = pd.DataFrame(
•   {
+   •   "A": 1.0,
•     "B": pd.Timestamp("20210101"),
•     "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
•     "D": np.array([12,5,6,9], dtype="int32"),
•     "E": pd.Categorical(["novo", "usado", "usado", "novo"])
•   }
+ )
+ print("Abaixo exibimos os dados da coluna C")
+ print(dataframe["C"])

print("Abaixo exibimos apenas um intervalo de linhas")
print(dataframe[0:2])

print("Abaixo exibimos apenas os valores do dataframe em que a coluna D seja maior do que 7")
print(dataframe[dataframe["D"] > 7])

print("Abaixo exibimos apenas os valores em que a coluna E contém o valor 'usado'")
print(dataframe[dataframe["E"].isin(["usado"])])
```

É possível fazer a exibição dos dados baseado em uma operação booleana. Nesse caso só serão exibidas linhas onde a coluna D tenha valores maiores do que 7



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)

print("Abaixo exibimos os dados da coluna C")
print(dataframe["C"])

print("Abaixo exibimos apenas um intervalo de linhas")
print(dataframe[0:2])

print("Abaixo exibimos apenas os valores do dataframe em que a coluna D seja maior do que 7")
print(dataframe[dataframe["D"] > 7])

print("Abaixo exibimos apenas os valores em que a coluna E contém o valor 'usado'")
print(dataframe[dataframe["E"].isin(["usado"])])
```

Também é possível exibir apenas as linhas que contenham valores em um determinado intervalo, através do método *isin*. Nesse caso só serão exibidas linhas em que a coluna E contém o valor “usado”.

Para a importação e exportação de dados, existem funções padrão no pandas.

Crie um arquivo chamado `arquivos_data_frame.py` para explorarmos algumas dessas operações.



```
• • • • • +
• • • import pandas as pd
• + • import numpy as np

+ • dataframe = pd.DataFrame(
    {
        • "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Exportando dataframe para o formato CSV")
dataframe.to_csv("arquivo_csv.csv")

print("Exportando dataframe para o formato XLSX")
dataframe.to_excel("arquivo_excel.xlsx", sheet_name="Sheet1")

print("Convertendo o dataframe para o formato JSON")
print(dataframe.to_json())
```

É possível exportar um dataframe para o formato csv.



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)

print("Exportando dataframe para o formato CSV")
dataframe.to_csv("arquivo_csv.csv")

print("Exportando dataframe para o formato XLSX")
dataframe.to_excel("arquivo_excel.xlsx", sheet_name="Sheet1")

print("Convertendo o dataframe para o formato JSON")
print(dataframe.to_json())
```

Também é possível exportar para o formato Excel. Para isso o pacote openpyxl deve estar instalado.



```
import pandas as pd
import numpy as np

dataframe = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20210101"),
        "C": pd.Series([1.2, 3.7, 5.5, 6], dtype="float32"),
        "D": np.array([12,5,6,9], dtype="int32"),
        "E": pd.Categorical(["novo", "usado", "usado", "novo"])
    }
)
print("Exportando dataframe para o formato CSV")
dataframe.to_csv("arquivo_csv.csv")

print("Exportando dataframe para o formato XLSX")
dataframe.to_excel("arquivo_excel.xlsx", sheet_name="Sheet1")

print("Convertendo o dataframe para o formato JSON")
print(dataframe.to_json())
```

A conversão para o formato JSON também é possível.



```
import pandas as pd
import numpy as np

dataframe = pd.read_csv("arquivo_csv.csv")
print("Aqui está o dataframe recuperado do arquivo CSV")
print(dataframe)

dataframe = pd.read_excel("arquivo_excel.xlsx")
print("Aqui está o dataframe recuperado do arquivo XLSX")
print(dataframe)
```

A abertura de arquivos e conversão para o formato dataframe é igualmente simples.

BIBLIOGRAFIA BÁSICA

BEAZLEY, David. *Python Essential Reference*, 2009.

BHARGAVA, ADITYA Y. *Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos*. São Paulo: Ed. Novatec, 2017

CORMEN, THOMAS H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2002

COSTA, Sérgio Souza. *Recursividade*. Professor Adjunto da Universidade Federal do Maranhão.

DOWNEY, ALLEN B. *Pense em Python. Pense como um cientista da computação*. São Paulo: Ed. Novatec, 2016

GRANATYR, Jones; PACHOLOK, Edson. *IA Expert Academy*. Disponível em: <https://iaexpert.academy/>

KOPEC, DAVID. *Problemas clássicos de ciência da computação com Python*. São Paulo: Ed. Novatec, 2019

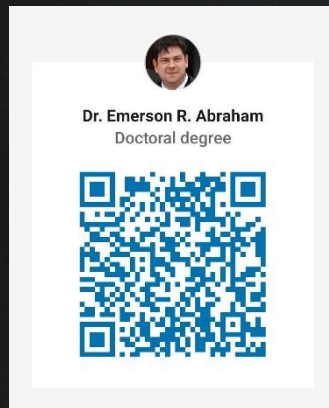
LINDEN, Ricardo. *Algoritmos Genéticos*. 3 edição. Rio de Janeiro: Editora Moderna, 2012

MCKINNEY, WILLIAM WESLEY. *Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython*. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. **Estrutura de Dados Usando C**. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. **Estruturas de Dados**. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. **Estruturas de Dados: Conceitos e Tecnicas de implementacao**. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. **Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java**. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estruturas de dados**. 2016.
- SILVA, Osmar Quirino. **Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações**. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. **Projeto de algoritmos com implementações em pascal e C**. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP

Copyright © 2023 | Professor Dr. Emerson R. Abraham

Todos os direitos reservados. A reprodução ou divulgação total ou parcial deste documento é expressamente proibida sem o consentimento formal, por escrito, do professor/autor.

