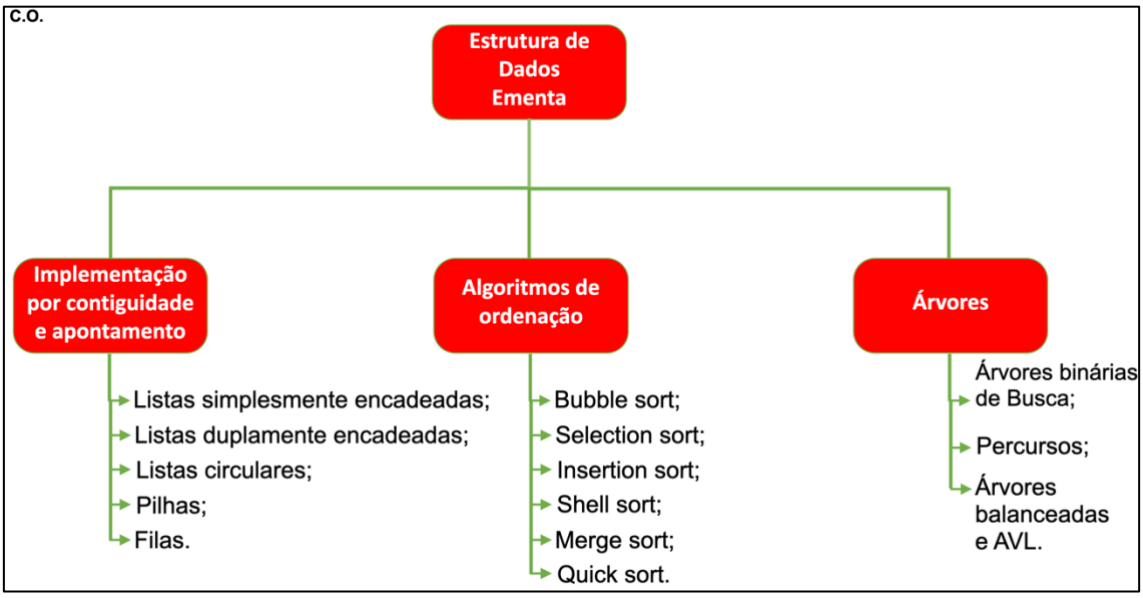
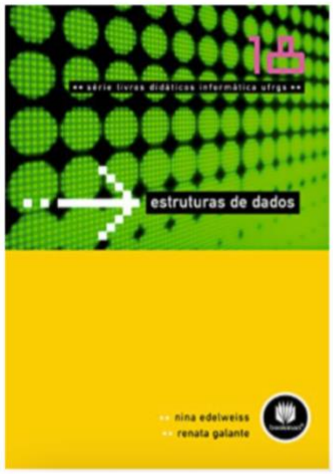
C.O.

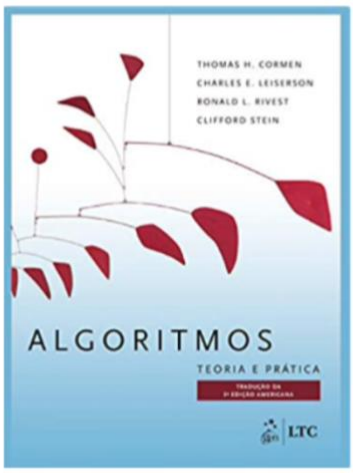


C.O.

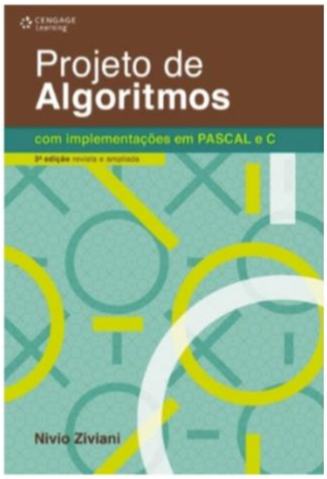
REFERENCIAS BIBLIOGRÁFICAS - EMENTA.



EDELWEISS, Nina; GALANTE, Renata. **Estruturas de dados**. Porto Alegre: Bookman, 2009.



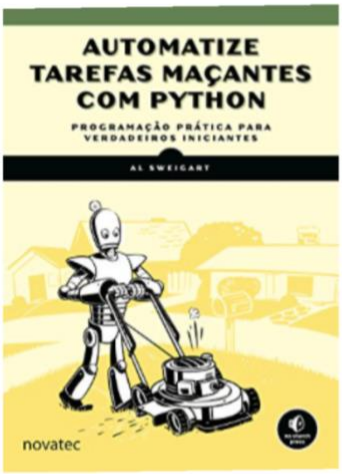
CORMEN, Thomas H. **Algoritmos: Teoria e Prática**. 3. ed. São Paulo: Elsevier, 2012



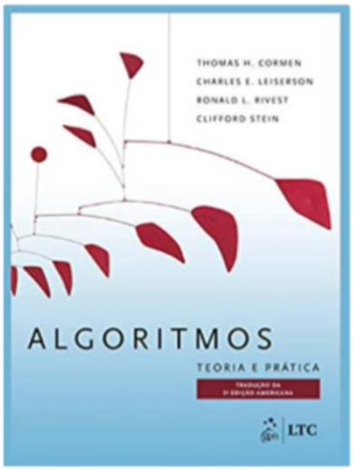
ZIVIANI, Nivio. **Projeto de algoritmos: com implementações em Pascal e C**. São Paulo, SP: Cengage Learning, 2015

C.O.

SUGESTÃO PROFESSOR



SWEIGART, Al. **Automatize tarefas maçantes com Python: Programação prática para verdadeiros iniciantes**. Novatec Editora. 2017



CORMEN, Thomas H. **Algoritmos: Teoria e Prática**. 3. ed. São Paulo: Elsevier, 2012

ROSSUM, **Guido Van. PEP 8 - Guia de Estilo Para Python**. Disponível em: <https://wiki.python.org.br/GuiaDeEstilo>

ALGORITMOS X ESTRUTURA DE DADOS

Segundo Ascencio & Araújo (2010), *apud* Cormen (2002) p.1, “um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída”.

Ao desenvolver um algoritmo, precisamos ficar atentos ao seu desempenho, pois, para executar uma função ele utilizará uma quantidade de memória e tempo do processador. Cada problema requer um algoritmo diferente e nossa tarefa será analisar o que pode ser mais eficiente levando em consideração os aspectos, como memória utilizada e tempo gasto para executá-lo (Ascencio & Araújo, 2010).

Um algoritmo manipula dados de entrada para gerar uma saída (resultado desejado), esses dados, quando são manipulados de forma homogênea, constituem um tipo **abstrato de dados**. Segundo Ascencio e Araújo (2010), *apud* (Pereira (1996), p. 1),

“...um tipo abstrato de dados é formado por um conjunto de valores e por uma série de funções que podem ser aplicadas sobre esses valores. Funções e valores, em conjunto, constituem um modelo matemático que pode ser empregado para 'modelar' e solucionar problemas do mundo real, especificando as características relevantes dos elementos envolvidos no problema, de que modo eles se relacionam e como podem ser manipulados”.

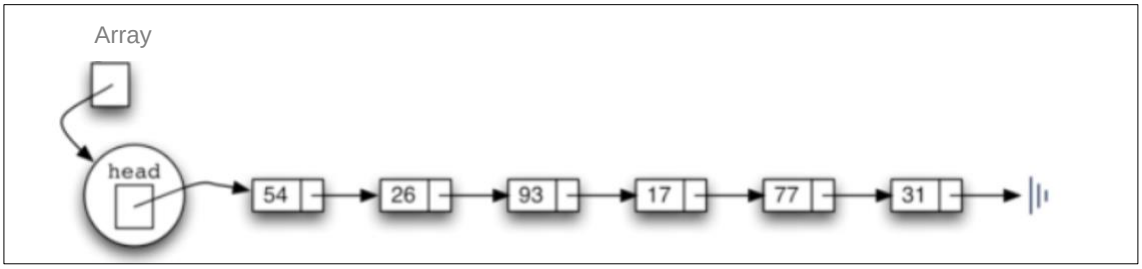
Um algoritmo é projetado em função de tipos abstratos de dados. Para implementá-los em uma linguagem de programação é necessário representá-los de alguma maneira nessa linguagem, utilizando tipos e operações suportadas pelo computador. Na representação do tipo abstrato de dados, emprega-se uma **estrutura de dados**” Ascencio e Araújo (2010).

“É durante o processo de implementação que a estrutura de armazenamento dos valores é especificada e os algoritmos que desempenharão o papel das funções (operações) são projetados” Ascencio e Araújo (2010).

Implementação de estrutura de dados por contiguidade: Segundo Thomas, Leiserson, Rivest, refere-se à ideia de armazenar elementos de uma estrutura de dados em locais de memória que estão lado a lado. Um exemplo

clássico disso são os **arrays** ou **vetores**, onde cada elemento é armazenado em uma posição adjacente na memória.

ARRAY: "Um array sempre armazena os dados em locais da memória sequencial, permitindo **tempos de acesso mais rápidos** em algumas situações, mas também diminuindo a velocidade de sua criação e a atualização geralmente é difícil.

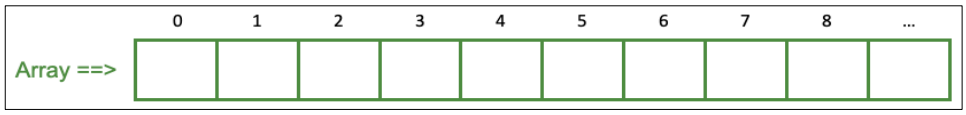


Fonte: adaptação do autor

Segundo o autor, quando temos volume grande de dados armazenados o array apresenta uma performance melhor que lista em Python. No exemplo a seguir importamos a biblioteca integrada array do Python para a criar estrutura do array:

 # Implementação de estrutura de dados por contiguidade

Cria um array vazio




```
from array import array

vetor = array('i',[ ])  # i = Type code
print(f'\nO array vetor contém os seguintes elementos: {vetor}')

print(f'\nO tipo de container que armazenou os dados é: {type(vetor)} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

O array vetor contém os seguintes elementos: array('i')

O tipo de container que armazenou os dados é: <class 'array.array'>
```

Retorna o typecode usado para criar o array.

```
print(f'\nO typecode usado para criar o array mumeros é: {vetor.typecode} \n')
```

```
O typecode usado para criar o array mumeros é: i
```

Segundo Ramalho (2021), typecode é um conceito relacionado a biblioteca array do Python. Cada typecode é uma string de um único caractere que especifica o tipo de elemento que o array irá conter.

O autor reforça que é importante entender que o typecode é essencial para a criação e gerenciamento de arrays. Em Python, pois, o typecode é quem

determina o tipo de dados que o array irá conter e, consequentemente a quantidade de memória que será alocada para cada elemento no array.

Typecode em Python:

'b' = inteiro com sinal (+ ou -) de 1 byte. Isso significa que ele pode armazenar números inteiros que vão de -128 a 127.

'B' = inteiro sem sinal de 1 byte. Isso significa que só pode representar números inteiros positivo no intervalo de 0 a 255;

'u' = caractere Unicode de 2 bytes, é usado para armazenar caracteres de texto em Unicode, que inclui praticamente todos os caracteres de todos os sistemas de escrita no mundo.

'h' = um inteiro com sinal de 2 bytes. Ele pode armazenar números inteiros de -32768 a 32767;

'H' = inteiro sem sinal de 2 bytes. Ele pode armazenar números inteiros de 0 a 65535;

'i' = inteiro com sinal de tamanho padrão, com tamanho padrão de 4 bytes, podendo variar de acordo com o sistema, aceita armazenar números -2147483648 a 2147483647;

'I' = inteiro sem sinal de tamanho padrão. O tamanho padrão é 4 bytes, e pode armazenar números de 0 a 4294967295.

'l' = inteiro longo com sinal, por padrão tem o mesmo tamanho que 'i';

'L' = inteiro longo sem sinal e segue o mesmo padrão de tamanho que 'I';

'f' = número de ponto flutuante de precisão simples, ou seja, usa 4 bytes e pode representar números com casas decimais.

'd' = número de ponto flutuante de precisão dupla. Ele usa 8 bytes e pode representar números com casas decimais com mais precisão do que 'f'.

Exemplos de uso de typecodes para criar arrays:

```
vetor_b = array('b', [-120, -90, 0, 90, 127])
print(f'\nb = inteiro com sinal de 1 byte: {vetor_b}')

vetor_B = array('B', [0, 60, 120, 180, 255])
print(f'\nB = inteiro com sinal de 1 byte: {vetor_B}')

vetor_u = array('u', 'Olá Mundo')
print(f'\nu: Caractere Unicode de 2 bytes: {vetor_u}')

vetor_h = array('h', [-30000, -10000, 0, 10000, 30000])
print(f'\nh = inteiro com sinal de 2 bytes: {vetor_h}')

vetor_H = array('H', [0, 10000, 20000, 30000, 65535])
print(f'\nH = inteiro sem sinal de 2 bytes: {vetor_H}')

vetor_i = array('i', [-2000000000, -1000000000, 0, 1000000000, 2000000000])
print(f'\ni = inteiro com sinal de tamanho padrão: {vetor_i}')
```

```
vetor_I = array('I', [0, 1000000000, 2000000000, 3000000000, 4294967295])
print(f'\nI = inteiro sem sinal de tamanho padrão: {vetor_I}')

vetor_l = array('l', [-2000000000, -1000000000, 0, 1000000000, 2000000000])
print(f'\nl = inteiro longo com sinal {vetor_l}')

vetor_L = array('L', [0, 1000000000, 2000000000, 3000000000, 4294967295])
print(f'\nl = inteiro longo sem sinal: {vetor_L}')

vetor_f = array('f', [1.0, 2.5, 3.5])
print(f'\nf = Ponto flutuante de precisão simples: {vetor_f}')

vetor_d = array('d', [1.0, 2.5, 3.5])
print(f'\nd = Ponto flutuante de precisão dupla: {vetor_d} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py
b = inteiro com sinal de 1 byte: array('b', [-120, -90, 0, 90, 127])
B = inteiro com sinal de 1 byte: array('B', [0, 60, 120, 180, 255])
u: Caractere Unicode de 2 bytes: array('u', 'Olá Mundo')
h = inteiro com sinal de 2 bytes: array('h', [-30000, -10000, 0, 10000, 30000])
H = inteiro sem sinal de 2 bytes: array('H', [0, 10000, 20000, 30000, 65535])
i = inteiro com sinal de tamanho padrão: array('i', [-2000000000, -1000000000, 0, 1000000000, 2000000000])
I = inteiro sem sinal de tamanho padrão: array('I', [0, 1000000000, 2000000000, 3000000000, 4294967295])
l = inteiro longo com sinal array('l', [-2000000000, -1000000000, 0, 1000000000, 2000000000])
l = inteiro longo sem sinal: array('L', [0, 1000000000, 2000000000, 3000000000, 4294967295])
f = Ponto flutuante de precisão simples: array('f', [1.0, 2.5, 3.5])
d = Ponto flutuante de precisão dupla: array('d', [1.0, 2.5, 3.5])
```

Populando o array vazio com a função for()

```
# Populando o array vazio com a função for()

from array import array

numeros = array('i', [])  # i = Type code

for n in range(3):
    numeros.append(int(input(f'\nDigite um número que irá para a memória na posição {n}: ')))

print(f'\n          ---------------------- \n')

print(f'\nO array numeros contém os seguintes elementos: {numeros}')

print(f'\nO tipo de container que armazenou os dados é: {type(numeros)} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py
Digite um número que irá para a memória na posição 0: 1
Digite um número que irá para a memória na posição 1: 3
Digite um número que irá para a memória na posição 2: 5

          ----------------------


O array numeros contém os seguintes elementos: array('i', [1, 3, 5])
O tipo de container que armazenou os dados é: <class 'array.array'>
```

Populando o array vazio com a função while

```
from array import array

letras = array('u', [])  # u = Type code

n = 0
while n < 3:
    letras.append(input(f'\nDigite um caractere que irá para a memória na posição {n}: '))
    n += 1

print(f'\n          ---------------------- \n')

print(f'\nO array letras contém os seguintes elementos: {letras}')

print(f'\nO tipo de container que armazenou os dados é: {type(letras)} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py
Digite um caractere que irá para a memória na posição 0: a
Digite um caractere que irá para a memória na posição 1: ç
Digite um caractere que irá para a memória na posição 2: ã
          ----------------------

O array letras contém os seguintes elementos: array('u', 'açã')
O tipo de container que armazenou os dados é: <class 'array.array'>
```

Observação: A biblioteca array do Python suporta apenas um caractere por posição de memória.

Populando o array vazio com nomes (strings) usando a função while e a biblioteca numpy

```
import numpy as np

nomes = np.array([], dtype='str')  # cria um array vazio de strings

n = 0
while n < 3:
    nome = input(f'\nDigite um nome que irá para a memória na posição {n}: ')
    nomes = np.append(nomes, nome)  # adiciona o nome ao array
    n += 1

print(f'\n          ---------------------- \n')

print(f'\nO array nomes contém os seguintes elementos: {nomes}')

print(f'\nO tipo de container que armazenou os dados é: {type(nomes)} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py
Digite um nome que irá para a memória na posição 0: Genoveva
Digite um nome que irá para a memória na posição 1: Gertrudez
Digite um nome que irá para a memória na posição 2: Gerimunda
          ----------------------

O array nomes contém os seguintes elementos: ['Genoveva' 'Gertrudez' 'Gerimunda']
O tipo de container que armazenou os dados é: <class 'numpy.ndarray'>
```

Observação: apesar do numpy ser mais eficiente com operações em arrays, seria um **exagero** utilizá-lo para amarzenar somente armazenando 3 nomes. Portanto, entenda que o exemplo acima serve para fins didáticos.

A biblioteca numpy oferece uma grande variedade de dtypes, que são usados para especificar o tipo de dados a ser armazenado em um array. Aqui estão alguns dos dtypes mais comuns:

int: Inteiro de tamanho padrão;
float: ponto flutuante de tamanho;
complex: número complexo representado por dois floats de 64 bits;
bool: booleano que armazena valores True e False;
object: objeto Python;
int64: número inteiro de 64 bits;
int32: número inteiro de 32 bits;
float64: número de ponto flutuante de 64 bits;
float32: búmero de ponto flutuante de 32 bits;
complex: número complexo representado por dois floats de 64 bits;
complex128: número complexo = dois floats de 64 bits;
complex64: número complexo = por dois floats de 32 bits;
str: Tipo de string;
unicode: string Unicode.

Principais diferenças entre os **dtypes** do numpy:

- Os nomes dos **dtypes em numpy** seguem um formato que indica o tipo de dado e o número de bits que cada valor do tipo ocupará na memória.

int32 e **int64:** são tipos de dados inteiros: int32 é um número inteiro que ocupa 32 bits de memória, enquanto int64 é um número inteiro que ocupa 64 bits de memória. Como int64 tem mais bits, ele pode representar uma faixa maior de números inteiros do que int32.

Int: quando aplicamos apenas Int como dtype, o numpy decidirá automaticamente o tamanho específico (int32, int64 etc.) baseado na plataforma que você está usando. Isso é chamado de tipo "*inteiro de plataforma*". Em um sistema operacional de 32 bits, "int" será equivalente a "int32", enquanto em um sistema de 64 bits, será equivalente a "int64".

float32 e **float64:** float32 é um número de ponto flutuante que ocupa 32 bits de memória, enquanto float64 é um número de ponto flutuante que ocupa 64 bits de memória. O float64 pode representar valores com maior precisão decimal do que float32 devido ao número maior de bits.

complex64, **complex128** e **complex**: são tipos de dados complexos. complex64 é um número complexo que ocupa 64 bits de memória, complex128

é um número complexo que ocupa 128 bits de memória, e complex é equivalente a complex128.

Lembre-se: Números complexos são um tipo de número que expande o conceito tradicional de números que conhecemos sobre os números reais ao adicionar uma parte imaginária. Um número complexo é geralmente escrito na forma a + bi, onde:

a: é a parte real do número;

b: é a parte imaginária do número;

i: é a unidade imaginária, que tem a propriedade $i^2 = -1$.

Exemplo: **3 + 4i** é um número complexo, onde 3 é a parte real e 4 é a parte imaginária.

Os números complexos são úteis em muitos campos da ciência e da engenharia, particularmente em situações que envolvem ondas ou oscilações, como a análise de circuitos elétricos ou o processamento de sinais.

É importante enteder que ao escolher o numpy com dtype, você precisa equilibrar as necessidades do seu programa entre a precisão dos dados e o uso eficiente da memória. Usar dtypes que ocupam mais bits resultará em maior precisão, mas também usará mais memória (MCKINNEY (2017), VANDERPLAS (2018), JOHANSSON (2018).

Exemplos de sintaxe usando diferentes dtypes do numpy:

```
import numpy as np

int_arr = np.array([1, 2, 3], dtype=int)
print(f'\nArray do tipo int: {int_arr}')
print(f'\n0 espaço em memória ocupado por um item do array do tipo int é: {int_arr.itemsize} bytes')

int32_arr = np.array([1, 2, 3], dtype=np.int32)
print(f'\nArray do tipo Int32: {int32_arr}')
print(f'\n0 espaço em memória ocupado por um item do array do tipo int32 é: {int32_arr.itemsize} bytes')

int64_arr = np.array([1, 2, 3], dtype=np.int64)
print(f'\nArray do tipo int64: {int64_arr}')
print(f'\n0 espaço em memória ocupado por um item do array do tipo int64 é: {int64_arr.itemsize} bytes')

float_arr = np.array([1.0, 2.0, 3.0], dtype=float)
print(f'\nArray do tipo Float: {float_arr}')
print(f'\n0 espaço em memória ocupado por um item do array do tipo Float é: {float_arr.itemsize} bytes')

float32_arr = np.array([1.0, 2.0, 3.0], dtype=np.float32)
print(f'\nArray do tipo Float32: {float32_arr}')
print(f'\n0 espaço em memória ocupado por um item do array do tipo Float32 é: {float32_arr.itemsize} bytes')

float64_arr = np.array([1.0, 2.0, 3.0], dtype=np.float64)
print(f'\nArray do tipo Float64: {float64_arr}')
print(f'\n0 espaço em memória ocupado por um item do array do tipo Float64 é: {float64_arr.itemsize} bytes')

bool_arr = np.array([True, False, True], dtype=bool)
print(f'\nArray do tipo Bool: {bool_arr}')
print(f'\n0 espaço em memória ocupado por um item do array tipo Bool é: {bool_arr.itemsize} bytes')
```

```
str_arr = np.array(['a', 'b', 'c'], dtype=str)
print(f'\nArray do tipo String array: {str_arr}')
print(f'\nO espaço em memória por um item do array tipo String é: {str_arr.itemsize} bytes')

unicode_arr = np.array(['a', 'b', 'c'], dtype=np.unicode_)
print(f'\nArray do tipo Unicode: {unicode_arr}')
print(f'\nO espaço em memória ocupado por um item do array tipo Unicode é: {unicode_arr.itemsize} bytes')

object_arr = np.array(['a', 'b', 'c'], dtype=object)
print(f'\nArray do tipo Object: {object_arr}')
print(f'\nO espaço em memória ocupado por um item do array do tipo Object é: {object_arr.itemsize} bytes')

complex_arr = np.array([1+2j, 2+3j, 3+4j], dtype=complex)
print(f'\nArray do tipo Complex: {complex_arr}')
print(f'\nO espaço em memória ocupado por um item do array do tipo Complex é: {complex_arr.itemsize} bytes')

complex64_arr = np.array([1+2j, 2+3j, 3+4j], dtype=np.complex64)
print(f'\nArray do tipo Complex64: {complex64_arr}')
print(f'\nO espaço em memória ocupado por um item do array do tipo Complex64 é: {complex64_arr.itemsize} bytes')

complex128_arr = np.array([1+2j, 2+3j, 3+4j], dtype=np.complex128)
print(f'\nArray do tipo Complex128 array: {complex128_arr}')
print(f'\nO espaço em memória ocupado por um item do array do tipo Complex128 é: {complex128_arr.itemsize} bytes \n')
```

```
Array do tipo Float64: [1. 2. 3.]

O espaço em memória ocupado por um item do array do tipo Float64 é: 8 bytes

Array do tipo Bool: [ True False  True]

O espaço em memória ocupado por um item do array tipo Bool é: 1 bytes

Array do tipo String array: ['a' 'b' 'c']

O espaço em memória por um item do array tipo String é: 4 bytes

Array do tipo Unicode: ['a' 'b' 'c']

O espaço em memória ocupado por um item do array tipo Unicode é: 4 bytes

Array do tipo Object: ['a' 'b' 'c']

O espaço em memória ocupado por um item do array do tipo Object é: 8 bytes

Array do tipo Complex: [1.+2.j 2.+3.j 3.+4.j]

O espaço em memória ocupado por um item do array do tipo Complex é: 16 bytes

Array do tipo Complex64: [1.+2.j 2.+3.j 3.+4.j]

O espaço em memória ocupado por um item do array do tipo Complex64 é: 8 bytes

Array do tipo Complex128 array: [1.+2.j 2.+3.j 3.+4.j]

O espaço em memória ocupado por um item do array do tipo Complex128 é: 16 bytes
```

Manipulando arrays com a biblioteca array Python

```
# Manipulando arrays com a biblioteca array Pyton

from array import array  # Importa o módulo array do Python

# Cria um array chamado vetor que contém elementos
vetor = array('i', [1, 2, 3, 4, 5])

# Imprime o valor do elemento) do array de acordo com a sua posição
print(f'\nO valor do elemento na posição zero do array é: {vetor[0]}')
print(f'\nO valor do elemento na posição dois do array é: {vetor[2]}')
```

```
# Deleta um elemento(s) do array conforme a posição definida
del vetor[1]

# Imprime os elementos que restaram no array
print(f'\nOs elementos que restarão no array são: {vetor}')

# Imprime o espaço ocupado de um elemento do array em bytes na memória
print(f'\nO espaço ocupado de um elemento do array na memória é: {vetor.itemsize} bytes')

# Imprime a quantidade de elementos que estão no array
print(f'\nA quantidade de elementos no array é: {len(vetor)}')

# Imprime o typecode do array
print(f'\nO typecode usado para criar o array é: {vetor.typecode}')

# Retorna o endereço de memória inicial do array e o tamanho do array
from array import array
vetor = array('u', ['a', 'b', 'a', 'é', 'a', 'ã', 'ç', 'a', 'z'])
print(f'\nO endereço de memória onde o array inicia e a quantidade de elementos do são: {vetor.buffer_info()}')

# Imprime o espaço ocupado em memória de todos os elementos do array  do tipo 'u' em bytes
print(f'\nO espaço total ocupado em memória pelo array tipo "u" em bytes é: {len(vetor) * vetor.itemsize}')

# Imprime o número de vezes que um valor se repete no array.
print(f'\nO número de vezes que a letra a ocorre no array é: {vetor.count("a")} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py
O valor do elemento na posição zero do array é: 1
O valor do elemento na posição dois do array é: 3
Os elementos que estão no array são: array('i', [1, 2, 3, 4, 5, 6])
Os elementos que restarão no array são: array('i', [1, 3, 4, 5, 6])
O espaço ocupado de um elemento do array na memória é: 4 bytes
A quantidade de elementos no array é: 5
O typecode usado para criar o array é: i
O endereço de memória onde o array inicia e a quantidade de elementos do são: (4408093712, 9)
O espaço total ocupado em memória pelo array tipo "u" em bytes é: 36
O número de vezes que a letra a ocorre no array é: 4
```

Manipulando arrays com numpy

```
import numpy as np

#  Cria um array vazio com número de elementos definido

vetor = np.empty (3, dtype=int) #  o número de elementos definido
print(f'\nO array criado possui {vetor} elementos')
print(f'\nO tipo do array criado é {type (vetor)}\n')


#  Cria um array vazio dinâmico

vetor = np.array([], dtype=int)
print(f'\nO array criado possui os seguintes elementos: {vetor}')
print(f'\nO tipo do array criado é {type (vetor)}\n')

# Adiciona um elemento no array dinâmico

vetor = np.append(vetor, 15)
print(f'\nO array criado possui os seguintes elementos: {vetor}')
print(f'\nO tipo do array criado é {type (vetor)}\n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py
O array criado possui [4613937818241073156 4611686018427387889 4607182418800017417] elementos
O tipo do array criado é <class 'numpy.ndarray'>

O array criado possui os seguintes elementos: []
O tipo do array criado é <class 'numpy.ndarray'>

O array criado possui os seguintes elementos: [15]
O tipo do array criado é <class 'numpy.ndarray'>
```

Matrizes em Python:

Estudamos que um array ou vetor é uma sequência de dados do mesmo tipo ocupando posições consecutivas de memória, tendo como vantagem a efeciência em trabalhar com um grande número volume da dados.

Na linguagem Python utilizamos o pacote NumPy para desenvolver estruturas de operações de arrays principalmente para criar matrizes. Uma matriz pode ser objeto do tipo: unidimensionais, bidimensionais e tridimensionais.

Uma matriz pode conter:

- Valores de um experimento com simulação em tempos discretos;
- Sinal gravado por um equipamento de medição como ondas sonoras;
- Pixels de uma imagem, escala de cinza ou coloridos;
- Dados tridimensionais medidos em posições X,Y,Z diferentes.

Na prática, raramente inserimos um por um os elementos de uma matriz, pois, normalmente elas são utilizadas nas seguintes áreas entre outras:

Ciência de Dados e Aprendizado de Máquina: são usadas para armazenar e manipular dados, fornecendo um meio eficiente de realizar operações em grandes conjuntos de dados, além de serem compatíveis com muitas bibliotecas de aprendizado de máquina, como scikit-learn, TensorFlow e PyTorch.

Computação Científica e Engenharia: são usadas em para representar sistemas de equações, transformações lineares etc.

Processamento de Imagem e Sinais: normalmente as imagens são reprentadas por matrizes multidimensionais de pixels, e a biblioteca numpy é aplicada para manipular essas matrizes em algoritmos de processamento de imagem.

Bioinformática: o numpy é usado para processar e analisar dados genômicos e proteômicos, que são frequentemente representados como matrizes.

Fonte: (mckinney (2017), Vanderplas (2018), Johansson (2018).

Manipulando matrizes com numpy

Matrizes de zero: são úteis em situações como:

Inicialização de Matrizes: É uma prática comum inicializar uma matriz com zeros e preenchê-la mais tarde com valores. Por exemplo, você pode não conhecer os valores no momento da criação da matriz, mas sabe que eles serão calculados posteriormente.

Placeholders* para Resultados Computados**: Você pode criar uma matriz de zeros para armazenar os resultados de algum cálculo. Esta matriz serviria como um ***placeholder para os valores que serão calculados.

Redefinição de uma Matriz Existente: Uma matriz de zeros pode ser usada para redefinir todos os elementos de uma matriz existente para zero.

Operações Matemáticas: caso onde seja preciso realizar operações matemáticas que envolvam uma matriz de zeros. Exemplo, a subtração de uma matriz de zeros de outra matriz resultará em uma matriz de sinais trocados.

Manipulação de Imagens: Em processamento de imagens, uma matriz de zeros pode representar uma imagem totalmente preta quando cada posição na matriz corresponde a um pixel. Fonte: (mckinney (2017), Vanderplas (2018), Johansson (2018).

Criando uma matriz de zeros

```
import numpy as np

matriz_0 = np.zeros ((3,4))
print (matriz_0)
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

  [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]
```

Criando uma matriz de zeros para manipular imagens

```
import numpy as np
from PIL import Image

# Cria a matriz de zeros, imagem preta
# 256 x 256 pixels, com 3 canais de cor (RGB)

matriz = np.zeros((256, 256, 3), dtype=np.uint8)

print(f'\n {matriz} \n')

# Cria uma imagem PIL a partir da matriz
img = Image.fromarray(matriz)

# Apresenta a a imagem na tela
img.show()
```

Matriz de uns: são úteis em situações semelhante à matriz de zeros:

Criando uma matriz de uns

```
import numpy as np

matriz_1 = np.ones ((3, 4), dtype=np.int16)
print(f'\n {matriz_1} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

  [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]
```

Criando matriz para aplicar em uma operação matemática:

```
import numpy as np

# Cria uma matriz 3x3 de números aleatórios
matriz = np.random.rand(3, 3)

print(f'\nMatriz de números aleatórios: \n')
print(matriz)

matriz_de_uns = np.ones((3, 3))

print(f'\nMatriz de uns')

print(f'\n {matriz_de_uns} \n')

# Adiciona o valor 5 a todos os elementos da matriz
matriz = matriz + 5*matriz_de_uns

print(f'\nMatriz original após adicionar o valor 5 \n')

print(f'{matriz} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

Matriz de números aleatórios:

[[0.24371583 0.78308522 0.81092795]
 [0.65001246 0.91628464 0.53240809]
 [0.88209197 0.76039623 0.76375533]]

Matriz de uns

 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]


Matriz original após adicionar o valor 5

[[5.24371583 5.78308522 5.81092795]
 [5.65001246 5.91628464 5.53240809]
 [5.88209197 5.76039623 5.76375533]]
```

Matriz com valores arranjados: Este tipo de matriz é aplica em uma vários contextos como:

Looping e Iteração: quando é preciso iterar sobre uma sequência de números, usamos este tipo de matriz para gerar esta sequência numérica;

Gráficos e Visualização de Dados: quando precisamos de uma matriz dos valores;

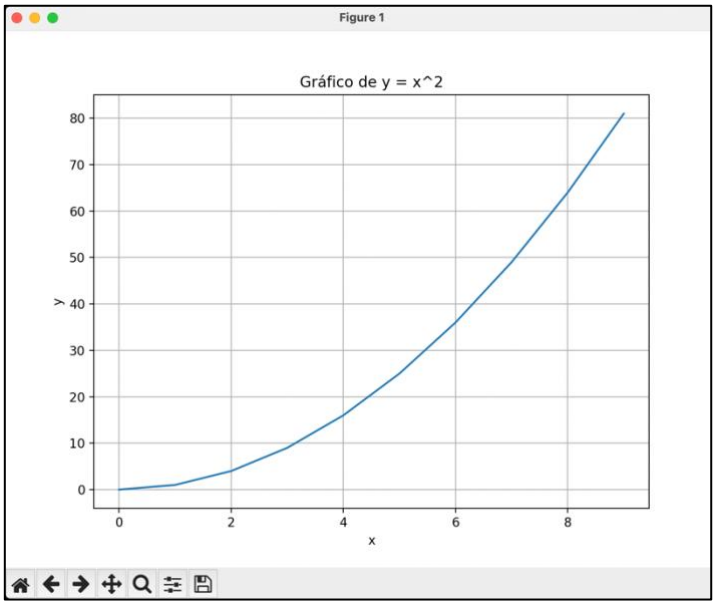
Cálculos Matemáticos: muito utilizado em matemática e ciência da computação, principalmente quando precisamos realizar cálculos em uma sequência numérica;

Testes e Análises: em cenários de teste, para testar um algoritmo ou função que tenham uma variedade de valores de entrada;

Criando uma matriz com valores arranjados:

```
x = np.arange(10)

print(f'\n {x} \n')

# Calcula y = x^2 para cada x
y = x**2

# Cria o gráfico
plt.figure(figsize=(8, 6))  # Define o tamanho da figura
plt.plot(x, y)  # Plota y contra x
plt.title('Gráfico de y = x^2')  # Define o título do gráfico
plt.xlabel('x')  # Define o rótulo do eixo x
plt.ylabel('y')  # Define o rótulo do eixo y
plt.grid(True)  # Adiciona um grid para facilitar a visualização
plt.show()  # Exibe o gráfico
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

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



Matriz de valores arranjados por número de amostras: é utilizada para representar uma sequência de números igualmente espaçados que serão utilizadas em aplicações como:

Gráficos e Visualização de Dados: usando por exemplo o matplotlib e outras bibliotecas de visualização de dados, numpy.linspace podemos gerar conjuntos de pontos para plotar funções.

Interpolação e Aproximação: o numpy.linspace é usado em análises numéricas e científicas para interpolação e aproximação de funções.

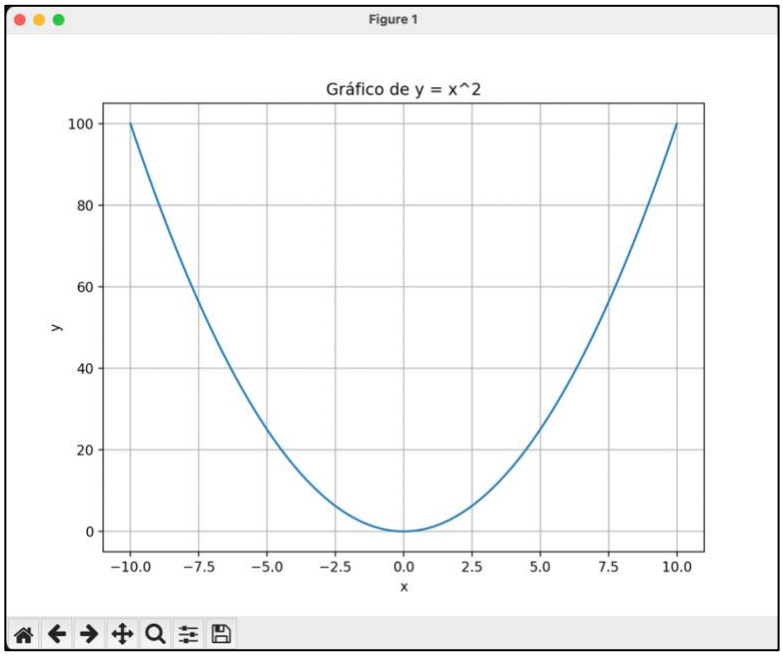
Simulação e Modelagem: tam'bem pode ser utilizado em simulações e modelagem, quando precisamos montar um conjunto de pontos de tempo ou espaço igualmente espaçados.

Testes e Análises: Em cenários de teste, quando for preciso testar um algoritmo ou função em uma variedade de valores de entrada

Criando uma matriz de valores arranjados por número de amostras utilizando o numpy, matplotlib e numpy.linspace para gerar um gráfico de visualização de dados:

```
import numpy as np
import matplotlib.pyplot as plt

# Gera 100 valores de x igualmente espaçados entre -10 e 10
x = np.linspace(-10, 10, 100)

print(f'\n Matriz com 100 valores de x igualmente espaçados entre -10 e 10')
print(f'\n {x} \n')

# Calcula y = x^2 para cada x
y = x**2

print(f'\n Matriz com equação calculada Calcula y = x^2 para cada x')
print(f'\n {x} \n')


# Cria o gráfico
plt.figure(figsize=(8, 6))  # Define o tamanho da figura
plt.plot(x, y)  # Plota y contra x
plt.title('Gráfico de y = x^2')  # Define o título do gráfico
plt.xlabel('x')  # Define o rótulo do eixo x
plt.ylabel('y')  # Define o rótulo do eixo y
plt.grid(True)  # Adiciona um grid para facilitar a visualização
plt.show()  # Exibe o gráfico
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

  Matriz com 100 valores de x igualmente espaçados entre -10 e 10

 [-10.          -9.7979798   -9.5959596   -9.39393939  -9.19191919
  -8.98989899  -8.78787879  -8.58585859  -8.38383838  -8.18181818
  -7.97979798  -7.77777778  -7.57575758  -7.37373737  -7.17171717
  -6.96969697  -6.76767677  -6.56565657  -6.36363636  -6.16161616
  -5.95959596  -5.75757576  -5.55555556  -5.35353535  -5.15151515
  -4.94949495  -4.74747475  -4.54545455  -4.34343434  -4.14141414
  -3.93939394  -3.73737374  -3.53535354  -3.33333333  -3.13131313
  -2.92929293  -2.72727273  -2.52525253  -2.32323232  -2.12121212
  -1.91919192  -1.71717172  -1.51515152  -1.31313131  -1.11111111
  -0.90909091  -0.70707071  -0.50505051  -0.3030303   -0.1010101
   0.1010101    0.3030303    0.50505051   0.70707071   0.90909091
   1.11111111   1.31313131   1.51515152   1.71717172   1.91919192
   2.12121212   2.32323232   2.52525253   2.72727273   2.92929293
   3.13131313   3.33333333   3.53535354   3.73737374   3.93939394
   4.14141414   4.34343434   4.54545455   4.74747475   4.94949495
   5.15151515   5.35353535   5.55555556   5.75757576   5.95959596
   6.16161616   6.36363636   6.56565657   6.76767677   6.96969697
   7.17171717   7.37373737   7.57575758   7.77777778   7.97979798
   8.18181818   8.38383838   8.58585859   8.78787879   8.98989899
   9.19191919   9.39393939   9.5959596    9.7979798   10.        ]


  Matriz com equação calculada Calcula y = x^2 para cada x

 [-10.          -9.7979798   -9.5959596   -9.39393939  -9.19191919
  -8.98989899  -8.78787879  -8.58585859  -8.38383838  -8.18181818
  -7.97979798  -7.77777778  -7.57575758  -7.37373737  -7.17171717
  -6.96969697  -6.76767677  -6.56565657  -6.36363636  -6.16161616
  -5.95959596  -5.75757576  -5.55555556  -5.35353535  -5.15151515
  -4.94949495  -4.74747475  -4.54545455  -4.34343434  -4.14141414
  -3.93939394  -3.73737374  -3.53535354  -3.33333333  -3.13131313
  -2.92929293  -2.72727273  -2.52525253  -2.32323232  -2.12121212
  -1.91919192  -1.71717172  -1.51515152  -1.31313131  -1.11111111
  -0.90909091  -0.70707071  -0.50505051  -0.3030303   -0.1010101
   0.1010101    0.3030303    0.50505051   0.70707071   0.90909091
   1.11111111   1.31313131   1.51515152   1.71717172   1.91919192
   2.12121212   2.32323232   2.52525253   2.72727273   2.92929293
   3.13131313   3.33333333   3.53535354   3.73737374   3.93939394
   4.14141414   4.34343434   4.54545455   4.74747475   4.94949495
   5.15151515   5.35353535   5.55555556   5.75757576   5.95959596
   6.16161616   6.36363636   6.56565657   6.76767677   6.96969697
   7.17171717   7.37373737   7.57575758   7.77777778   7.97979798
   8.18181818   8.38383838   8.58585859   8.78787879   8.98989899
   9.19191919   9.39393939   9.5959596    9.7979798   10.        ]
```



Diferenças entre uma matriz com **valores arranjados** e uma matriz de de **valores arranjados por número de amostras:** e uma matriz com valores arranjados gerada por numpy.arange e uma matriz de valores arranjados por número de amostras gerada por numpy.linspace está na maneira como os valores na matriz são gerados e espaçados.

- O **numpy.arange** gera uma matriz de valores espaçados uniformemente dentro de um intervalo dado. Exemplo: numpy.arange(0, 10, 2) ele irá gerar uma matriz de números inteiros pares entre 0 e 10.

- O **numpy.linspace** gera uma matriz de valores igualmente espaçados sobre um intervalo especificado, com base no número de pontos que você deseja. Exemplo, numpy.linspace(0, 10, 5) gera uma matriz de 5 números que são igualmente espaçados entre 0 e 10 (inclusive). Fonte: (mckinney (2017), Vanderplas (2018), Johansson (2018).

Matriz com os mesmos números – constantes: são úteis em vários cenários. Exemplos:

Inicialização: usado em algoritmos de aprendizado de máquina e otimização, precisamos inicializar os parâmetros ou variáveis com um valor constante;

Cálculos matriciais: usado em cálculos de álgebra linear, principalmente quando precisamos adicionar, subtrair ou fazer outras operações com uma matriz constante. numpy.full permite que você crie essa matriz constante;

Processamento de imagens: quando precisamos criar uma imagem de um único valor ou cor.

- Na matriz constante do exemplo segue aplicamos a função numpy.full((2, 3), 7) que pode ser utilizada em vários contextos, pois, a função gera uma matriz 2x3 onde todos os elementos são 7.

- Vamos aplicá-la no seguinte cenário: Considere que estamos trabalhando com um conjunto de dados e precisamos que este conjunto tenha uma operação que precise subtrair 7 de todos os elementos de outra matriz. Nesse caso, vamos utilizar a matriz que criamos para realizar essa operação Fonte: (mckinney (2017), Vanderplas (2018), Johansson (2018).

Criando uma matriz com os mesmos números – constantes:

```
import numpy as np

# Matriz constante
matriz_c = np.full((2, 3), 7)
print(f'\nMatriz constante \n')
print(f'{matriz_c} \n')

# Matriz aleatória
matriz_random = np.random.rand(2,3) * 10
print(f'\nMatriz aleatória \n')
print(f'{matriz_random} \n')

# Subtraindo 7 de todos os elementos usando a matriz constante
matriz_resultado = matriz_random - matriz_c

print(f'\nMatriz após a subtração \n')
print(f'{matriz_resultado} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

Matriz constante

[[7 7 7]
 [7 7 7]]


Matriz aleatória

[[3.7464793  3.92768925 0.17580705]
 [7.31694848 6.39981718 5.02417527]]


Matriz após a subtração

[[-3.2535207  -3.07231075 -6.82419295]
 [ 0.31694848 -0.60018282 -1.97582473]]
```

Matriz de identidade com zeros e uns: é uma matriz que possui o mesmo número de linhas e colunas e é poupulada com 1s na diagonal principal e 0s em todos os outros lugares. Ela recebe esse nome porque, na álgebra linear, multiplicar qualquer matriz por uma matriz de identidade resulta na mesma matriz original, assim como multiplicar qualquer número por 1 resulta no mesmo número original. Algumas aplicações:

- **Álgebra Linear:** por ser um elemento neutro para a operação de multiplicação de matrizes, quando qualquer matriz é multiplicada pela matriz de identidade, a matriz original é retornada;

Transformações Lineares: representam transformações lineares que não alteram o espaço vetorial. É muitos utilizada nos campos da computação gráfica e aprendizado de máquina;

Inversão de Matriz: usada no processo de encontrar a matriz inversa.

Cálculos de Autovalores e Autovetores: esses são conceitos fundamentais em campos como a física quântica e a aprendizagem de máquina.

Criando uma matríz identidade com zeros e uns:

```
import numpy as np

# Cria uma matriz de identidade 3x3
matriz_identidade = np.eye(3)

print(f'\nMatriz identidade com zeros e uns \n')
print(f'{matriz_identidade} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

Matriz identidade com zeros e uns

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Matriz com valores Diagonais: são utilizadas principalmente em contextos matemáticos e computacionais. Além disso essas matrizes são mais fáceis de inverter e resolver do que outras matrizes mais gerais. Exemplo:

Criando uma matriz com valores diagonais:

```
import numpy as np
import matplotlib.pyplot as plt

print(f'\nMatriz com valores diagonais \n')
matriz_t = np.diag(np.array([1, 2, 3, 4]))
print(f'{matriz_t} \n')

vetor = np.random.rand(4)
print(f'\nVetor aleatório \n')
print(f'{vetor} \n')

resultado = np.dot(matriz_t, vetor)
print(f'\nMultiplicação da matriz pelo vetor \n')
print(f'{resultado} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 001_aula.py

Matriz com valores diagonais

[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]


Vetor aleatório

[0.33380196 0.03860259 0.26745635 0.01009546]


Multiplicação da matriz pelo vetor

[0.33380196 0.07720518 0.80236905 0.04038186]
```

No exemplo as matrizes diagonais podem ser usadas para realizar operações de escala em vetores e outras matrizes.

Matriz unidirecional para gerar gráficos 2D: são utilizados em disciplinas e campos de estudo para visualizar dados e extrair *insights*. São conseirados ferramentas fundamentais para a análise de dados. Exemplos de onde podem ser usados:

Ciência e Engenharia: são usados para representar dados experimentais e teóricos. Por exemplo, um físico pode usar um gráfico 2D para traçar a posição de um objeto em função do tempo, enquanto um engenheiro pode usá-lo para representar a relação entre a tensão e a deformação de um material.

Eonomia e Negócios: podem ser usados para visualizar tendências ao longo do tempo, como as mudanças nos preços das ações, taxas de inflação, ou vendas de um produto particular.

Medicina e Saúde: podem representar relações entre diferentes medidas de saúde, como a relação entre idade e peso, ou a progressão de uma doença em diferentes pacientes.

Geografia e Ciências Ambientais: podem ser usados para mostrar mudanças em características geográficas ou ambientais, como o nível do mar em relação ao tempo ou a concentração de um poluente em diferentes locais.

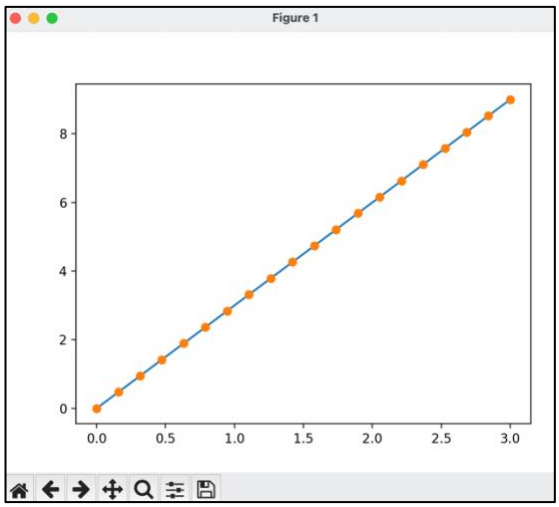
Educação: são utilizados como ferramentas de ensino valiosas em matemática e ciências para ajudar os alunos a entenderem conceitos como funções, relações lineares e não lineares.

Controle de Qualidade e Manufatura: Na indústria são usados para monitorar a qualidade de produtos e processos, traçando variáveis como tolerâncias de fabricação, defeitos, ou eficiência de produção.

Aprendizado de Máquina e Análise de Dados: são usados para visualizar dados, encontrar padrões, identificar clusters, ou para avaliar o ajuste de modelos (mckinney (2017), Vanderplas (2018), Johansson (2018).

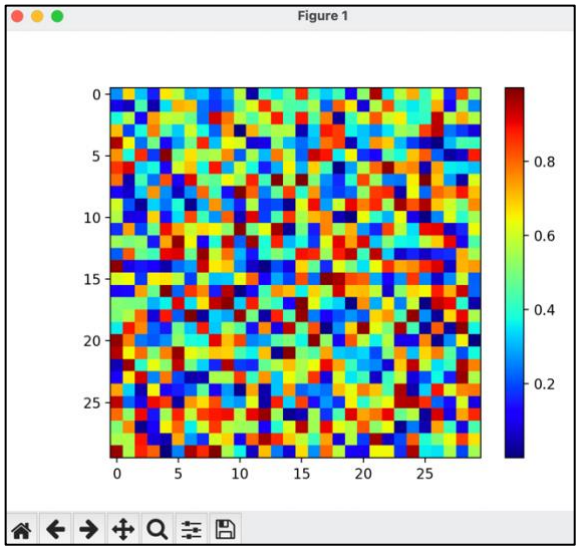
Criando um gráfico 2D – matriz unidirecional:

```
import numpy as np
import matplotlib.pyplot as plt

# Gera os dados para os eixos x e y
eixo_x = np.linspace(0, 3, 20)
eixo_y = np.linspace(0, 9, 20)

# Cria um gráfico de linha
plt.plot(eixo_x, eixo_y)

# Cria um gráfico de pontos
plt.plot(eixo_x, eixo_y, 'o')

# Exibe o gráfico
plt.show()
```



Criando um gráfico 2D – matriz bidirecional – ruídos aletórios / mapa de calor:

```
import numpy as np

# Visualização - bidimensional
import matplotlib.pyplot as plt
image = np. random. rand (30, 30)
plt. imshow (image, cmap=plt.cm. jet)
plt. colorbar ()
plt. show ()
```



No exemplo acima criamos uma imagem bidimensional e a apresentamos usando a biblioteca matplotlib.

- Primeiro, criamos uma matriz 30x30 preenchida com números aleatórios entre 0 e 1 usando a função numpy.random.rand;

- Depois, usamos a função matplotlib.pyplot.imshow para exibir esta matriz como uma imagem;

- O argumento cmap=plt.cm.jet indica estamos usando o mapa de cores 'jet', que é uma gama de cores que vai do azul escuro (para valores baixos) até o vermelho (para valores altos). Dessa maneira para cada valor na matriz foi mapeado para uma cor correspondente no mapa de cores;

- A função matplotlib.pyplot.colorbar adicionou uma barra de cores ao lado da imagem, indicando quais cores correspondem a quais valores;

- O matplotlib.pyplot.show foi usado para exibir a imagem.

É importante saber que esta técnica de visualização é muito utilizada em física e engenharia, onde, imagens semelhantes a esta podem ser usadas para visualizar campos de temperatura, pressão, densidade, ou qualquer outra quantidade que possa ser representada em uma grade bidimensional.

- Em ciência de dados e aprendizado de máquina, isso pode ser útil para visualizar matrizes de covariância, mapas de calor, ou outras representações de dados em duas dimensões;

- Por fim, é importante ressaltar apesar do estar gerando um tipo de mapa de calor, note que os "**ruídos**" apresentados são o resultado de números aleatórios, e não representam qualquer tipo de dado real ou tendência significativa mckinney (2017), Vanderplas (2018), Johansson (2018).

Matrizes e gráficos tridimensionais: alguns exemplos de utilização:

Medicina: Em exames de imagem médica como tomografias computadorizadas (TC) e ressonâncias magnéticas (RM), os dados são frequentemente representados como uma matriz tridimensional de valores de intensidade de pixel. Isso permite aos médicos visualizar estruturas internas do corpo em três dimensões.

Geociências: Em geologia e exploração de petróleo, a visualização 3D é usada para representar dados geofísicos e geológicos. Isso pode incluir dados de sismografia, que são usados para criar imagens 3D do subsolo.

Engenharia e Design: Em engenharia mecânica e design industrial, a visualização 3D é frequentemente usada para modelar peças e sistemas. Isso permite que os engenheiros visualizem e testem o design antes da produção.

Meteorologia: Na meteorologia, modelos tridimensionais do clima são usados para prever o tempo e estudar fenômenos climáticos.

Física e Química: Em física e química, a visualização 3D é usada para representar campos elétricos, magnéticos, campos de gravidade, estruturas moleculares, entre outros.

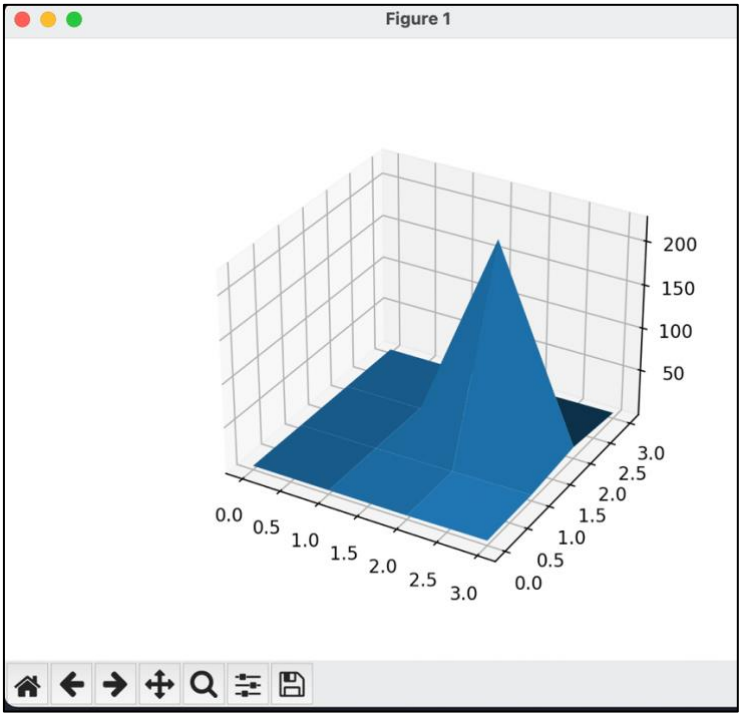
Ciência de Dados e Aprendizado de Máquina: Em ciência de dados, a visualização 3D pode ser usada para visualizar dados com três ou mais variáveis, para entender melhor a estrutura e os padrões nos dados.

Astronomia: Em astronomia, a visualização 3D é usada para representar dados sobre estrelas, galáxias e outros fenômenos espaciais.

É importante tem em mente que, embora a visualização 3D ser muito útil, ela apresenta uma alta complexidade o que a torna mais difícil de trabalhar do que a visualização 2D, devido à dificuldade adicional de representar um objeto tridimensional em uma tela bidimensional mckinney (2017), Vanderplas (2018), Johansson (2018).

Criando uma matriz e gráfico tridimensional

```python
# Importa as bibliotecas necessárias
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Cria a matriz 2D
# Cada lista dentro da lista principal representa uma linha na matriz
matriz = np.array([
    [1, 1, 1, 2],
    [1, 1, 2, 3.33],
    [1, 1, 223, 10],
    [1, 1, 2, 3]
])

# Define as dimensões da matriz - neste caso, as linhas e as colunas
rows = range(matriz.shape[0])
columns = range(matriz.shape[1])

# Cria uma figura para o gráfico
fig = plt.figure()

# Adiciona um subplot à figura com uma projeção 3D
ax = fig.add_subplot(111, projection="3d")

# Cria uma grade de coordenadas a partir das dimensões da matriz
X, Y = np.meshgrid(rows, columns)

# Plota a superfície 3D usando as coordenadas da grade e os valores da matriz
ax.plot_surface(X, Y, matriz)

# Exibe o gráfico
plt.show()
```



Atividade de postagem: Poste no ambiente virtual.ifro.edu.br a tarefa: "Poste aqui em um único arquivo.py todos os códigos exemplos apresentados no material de 02-08-2023 até o dia 09-08-2023 às 18:30".

Referências:

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela dos Santos de. **Estruturas de dados : algoritmos, análise da complexidade e implementações.** São Paulo : Pearson Prentice Hall, 2010. ISBN 978-85-7605-881-6

CORMEN, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms". MIT Press, 3rd Edition, 2009.

RAMALHO, Luciano. **Fluent Python: Clear, Concise, and Effective Programming".** Editora O'Reilly Media. Agosto de 2021

MCKINNEY, W. **Python para Análise de Dados: Tratamento de Dados com Pandas, NumPy e IPython. 2. ed**. São Francisco: O'Reilly Media, 2017.

VANDERPLAS, **Python Data Science Handbook: Essential Tools for Working with Data. São Francisco:** O'Reilly Media, 2016.

JOHANSSON, **Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib (2ª ed.).** Nova Iorque: Apress, 2018.