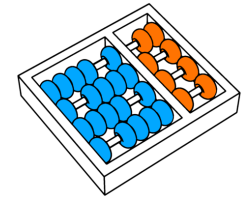



# Introdução ao Processamento Digital de Imagem
# MC920 / MO443

Prof. Hélio Pedrini

Instituto de Computação

UNICAMP

http://www.ic.unicamp.br/~helio

1º Semestre de 2024

# Linguagem de Programação Python

- Python é uma linguagem de programação de alto nível de abstração para uso geral desenvolvida por Guido van Rossum e lançada em 1991.
- Vários pacotes oferecem uma ampla gama de funcionalidades, tais como:
  - interfaces gráficas de usuário
  - processamento de texto
  - administração de sistemas
  - processamento de imagens
  - computação científica
  - aprendizado de máquina
  - bancos de dados
  - visão computacional

# Linguagem de Programação Python

- NumPy: biblioteca otimizada para operações numéricas, incluindo suporte para grandes vetores e matrizes multidimensionais.
- SciPy: biblioteca para computação científica, que inclui pacotes para algoritmos de otimização, rotinas de álgebra linear, integração numérica, ferramentas de interpolação, funções estatísticas, ferramentas de processamento de sinal.
- Matplotlib: biblioteca para geração de gráficos, histogramas, gráficos de barras, gráficos de dispersão, entre vários outros recursos gráficos.
- OpenCV: biblioteca que inclui algoritmos de visão computacional e processamento de imagens usados para detecção e reconhecimento facial, rastreamento de objetos, calibração de câmeras, geração de nuvem de pontos a partir de câmeras estéreo, entre muitos outros.
- Scikit-Image: biblioteca que contém uma coleção de algoritmos de processamento de imagens.
- Scikit-Learn: biblioteca que implementa vários algoritmos de aprendizado de máquina e ferramentas para análise de dados.
- Pandas: biblioteca que fornece estruturas de dados, ferramentas de análise de dados e operações para manipulação de tabelas numéricas e séries temporais.

# Fontes de Referência

- Python: `https://docs.python.org/`
- NumPy: `http://www.numpy.org/`
- SciPy: `https://www.scipy.org/`
- Matplolib: `http://matplotlib.org/`
- OpenCV: `http://docs.opencv.org/`
- Scikit-Image: `http://scikit-image.org/`
- Scikit-Learn: `http://scikit-learn.org/`
- Pandas: `http://pandas.pydata.org/`

# Modos de Programação

Python possui dois modos de programação:

- *interativo*
- *script*

# Modo Interativo

Chama o interpretador sem passar parâmetros:

```
$ python
```

Saída:

```
Python 3.11.2 (default, Mar 31 2023, 17:51:05)
[GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Exemplo:

```
>>> print("Hello, World!")
```

Saída:

```
Hello, World!
```

# Modo Script

Chama o interpretador com programa a ser executado (assume-se que o código a seguir está incluído no arquivo "prog.py"):

```
print("Hello, World!")
```

Exemplo:

```
$ python prog.py
```

Saída:

```
Hello, World!
```

# Indentação

- Blocos de código são denotados pela indentação das linhas.
- Embora o número de espaços na indentação seja variável, todas as declarações dentro do bloco devem ser indentadas na mesma quantidade.

Correto:

```
if True:
   print("True")
else:
   print("False")
```

Incorreto:

```
if True:
   print("Answer")
   print("True")
else:
   print("Answer")
  print("False")
```

# Citações

- Python aceita aspas simples (') e duplas (") para denotar literais de cadeias de caracteres, desde que o mesmo tipo de aspas comece e termine a cadeia de caracteres.

Exemplos:

```
palavra = 'palavra'
frase = "exemplo de frase"
```

# Comentários

- O caractere '#' que não está no interior de um literal de cadeia de caracteres inicia um comentário.
- Todos os caracteres após o '#' e até o final da linha fazem parte do comentário e o interpretador Python os ignora.

Exemplo:

```
# primeiro comentário
print("Hello, World!") # segundo comentário
```

# Atribuição de Valores a Variáveis

- As variáveis em Python não precisam de declaração explícita para reservar espaço na memória.
- A declaração ocorre automaticamente quando um valor é atribuído a uma variável. O operador de atribuição é o sinal de igual '='.

Exemplo:

```
contador = 50        # inteiro
milhas   = 7000.0    # ponto flutuante
nome     = "Maria"   # cadeia de caracteres

print(contador)
print(milhas)
print(nome)
```

# Múltiplas Atribuições

Exemplo:

```
a = b = c = 0
x, y, z = "Laranja", "Banana", "Cereja"
```

# Tipos Básicos de Dados

- Números
- Cadeias de Caracteres
- Listas
- Tuplas
- Dicionários

# Tipos Numéricos

- `int`: inteiros com sinal
- `long`: inteiros longos (também podem ser representados em octal e hexadecimal)
- `float`: valores em ponto flutuante
- `complex`: números complexos

# Cadeias de Caracteres

- Uma cadeia de caracteres é um conjunto contíguo de caracteres delimitados entre aspas simples ou duplas.
- Subcadeias de caracteres podem ser extraídas com o operador de indexação '[ ]' e o operador de fatiamento '[:]', com os índices iniciando em 0 para o primeiro caractere na cadeia e $-1$ representando o último caractere.
- Operadores associados: concatenação ('+') e repetição ('*').

Exemplo:

```
str = 'Hello, World!'

print(str)          # imprime cadeia completa
print(str[0])       # imprime primeiro caractere da cadeia
print(str[2:5])     # imprime terceiro ao quinto caractere
print(str[2:])      # imprime cadeia iniciando do terceiro caractere
print(str * 2)      # imprime cadeia duas vezes
print(str + "!!") # imprime cadeia concatenada
```

Saída:

```
Hello, World!
H
llo
llo, World!
Hello, World!Hello, World!
Hello, World!!!
```

# Listas

- Uma lista contém itens separados por vírgulas e envolvidos entre colchetes '[ ]'.
- Os itens pertencentes a uma lista podem ser de diferentes tipos de dados.
- O acesso aos elementos de uma lista é realizado de forma similar ao acesso às cadeias de caracteres.
- Operadores associados: concatenação ('+') e repetição ('*').

Exemplo:

```
lista1 = ['maria', 258, 13.67, 'texto', 12]
lista2 = [158, 'maria']

print(lista1)           # imprime a lista completa
print(lista1[0])        # imprime o primeiro elemento da lista
print(lista1[1:3])      # imprime segundo ao terceiro caractere
print(lista1[2:])       # imprime os elementos a partir do terceiro elemento
print(lista2 * 2)       # imprime a lista duas vezes
print(lista1 + lista2) # imprime as listas concatenadas
```

Saída:

```
['maria', 258, 13.67, 'texto', 12]
maria
[258, 13.67]
[13.67, 'texto', 12]
[158, 'maria', 158, 'maria']
['maria', 258, 13.67, 'texto', 12, 158, 'maria']
```

# Tuplas

- Uma tupla contém itens separados por vírgulas e envolvidos entre parênteses '( )'.
- Diferentemente das listas, as tuplas não podem ser atualizadas (somente para leitura).
- Operadores associados: concatenação ('+') e repetição ('*').

Exemplo:

```
tupla1 = ('maria', 258, 13.67, 'texto', 12)
tupla2 = (158, 'maria')

print(tupla1)           # imprime a lista completa
print(tupla1[0])        # imprime o primeiro elemento da lista
print(tupla1[1:3])      # imprime segundo ao terceiro elemento
print(tupla1[2:])       # imprime os elementos a partir do terceiro elemento
print(tupla2 * 2)       # imprime a lista duas vezes
print(tupla1 + tupla2) # imprime as listas concatenadas
```

Saída:

```
('maria', 258, 13.67, 'texto', 12)
maria
(258, 13.67)
(13.67, 'texto', 12)
(158, 'maria', 158, 'maria')
('maria', 258, 13.67, 'texto', 12, 158, 'maria')
```

# Dicionários

- Dicionários consistem em pares de chave-valor, os quais são indexados por chaves únicas (normalmente números ou cadeias de caracteres). Um valor é associado a cada chave.
- Dicionários são delimitados por chaves '{ }', e os valores podem ser atribuídos e acessados usando colchetes '[]'.

Exemplo:

```
dic1 = {}
dic1['um'] = "Chave um"
dic1[2] = "Chave dois"
dic2 = {'nome': 'pedro', 'codigo': 6734, 'departamento': 'vendas'}

print(dic1['um'])     # imprime valor para a chave 'um'
print(dic1[2])        # imprime valor para a chave 2
print(dic2)           # imprime dicionário completo
print(dic2.keys())    # imprime todas as chaves
print(dic2.values()) # imprime todos os valores
```

Saída:

```
Chave um
Chave dois
{'nome': 'pedro', 'codigo': 6734, 'departamento': 'vendas'}
['nome', 'codigo', 'departamento']
['pero', 6734, 'vendas']
```

# Operadores Aritméticos

`+  -  *  /  %  **  //`

Exemplo:

```
print(5*7)
```

Saída:

```
35
```

Exemplo:

```
print(3**2)
```

Saída:

```
9
```

Exemplo:

```
print(6.0 // 3.5)
```

Saída:

```
1.0
```

# Operadores Relacionais

`==` `!=` `>` `<` `>=` `<=`

Exemplo:

```
print(7 > 10)
```

Saída:

```
False
```

Exemplo:

```
print(4 == 4)
```

Saída:

```
True
```

# Operadores de Atribuição

`=  +=  -=  *=  /=  %=  **=  //=`

Exemplo:

```
a = 5
a *= 3
print(a)
```

Saída:

```
15
```

# Operadores Lógicos

`and  or  not`

Exemplo:

```
idade = 32
salario = 35000
print((idade > 25) and (salario > 40000))
```

Saída:

```
False
```

# Operadores Bitwise

`&  |  ^  ~  <<  >>`

Exemplo:

```
a = 00111100    # decimal 60
b = 00001101    # decimal 13

print(a & b)
print(a | b)
print(a ^ b)
print(~a)
```

Saída:

```
00001100    # decimal 12
00111101    # decimal 61
00110001    # decimal 49
11000011    # decimal -61
```

Exemplo: Testa se número 'x' é par ou ímpar.

# Operadores Bitwise

```
&   |   ^   ~   <<   >>
```

Exemplo:

```
a = 00111100    # decimal 60
b = 00001101    # decimal 13

print(a & b)
print(a | b)
print(a ^ b)
print(~a)
```

Saída:

```
00001100    # decimal 12
00111101    # decimal 61
00110001    # decimal 49
11000011    # decimal -61
```

Exemplo: Testa se número 'x' é par ou ímpar.

```
x = int(input('Entre um número: '))
print('x é %s.' % ('par', 'ímpar')[x&1])
```

# Operadores de Pertencimento

Os operadores de pertencimento testam a pertença em uma sequência, como cadeias de caracteres, listas ou tuplas.

`in` `not in`

Exemplo:

```
s = 'Cadeia de Caracteres'
for c in s:
    print(c, end=" ")
print('p' in s)
print(' ' in s)
print('b' in s)
```

Saída:

```
C a d e i a   d e   C a r a c t e r e s
False
True
False
```

# Operadores de Pertencimento

Exemplo:

```
a = 10
b = 2
lista = [1, 2, 3, 4, 5];

if (a in lista):
   print("a pertence à lista")
else:
   print("a não pertence à lista")

if (b not in lista):
   print("b não pertence à lista")
else:
   print("b pertence à lista")
```

Saída:

```
a não pertence à lista
b pertence à lista
```

# Operadores de Identidade

Operadores de identidade comparam posições de memória de dois objetos.

`is` `is not`

Exemplo:

```
a = 20
b = 20

if a is b:
   print("a e b têm mesma identidade")
else:
   print("a e b não têm mesma identidade")

if id(a) == id(b):
   print("a e b têm mesma identidade")
else:
   print("a e b não têm mesma identidade")

b = 30
if a is b:
   print("a e b têm mesma identidade")
else:
   print("a e b não têm mesma identidade")
```

Saída:

```
a e b têm mesma identidade
a e b têm mesma identidade
a e b não têm mesma identidade
```

# Comandos Condicionais

Exemplo:

```
nota = float(input("Entre nota: "))

if nota >= 50:
   print('Aprovado')
else:
   print('Reprovado')
```

Exemplo:

```
nota = float(input("Entre nota: "))

if nota >= 90:
   conceito = 'A'
else:
   if nota >= 80:
      conceito = 'B'
   else:
      if nota >= 70:
         conceito = 'C'
      else:
         if nota >= 60:
            conceito = 'D'
         else:
            conceito = 'F'
print('\nConceito: %s' % conceito)
```

# Comandos Condicionais

Exemplo:

```
nota = float(input("Entre nota: "))

if nota >= 90:
   conceito = 'A'
elif nota >= 80:
   conceito = 'B'
elif nota >= 70:
   conceito = 'C'
elif nota >= 60:
   conceito = 'D'
else:
   conceito = 'F'
print('\nConceito: %s' % conceito)
```

# Comando de Repetição (*While*)

Repete um comando ou bloco de comandos enquanto uma determinada condição for verdadeira. A condição é avaliada antes de executar o corpo do laço.

Exemplo:

```
total = 0.0
contador = 0
while contador < 10:
    numero = float(input("Entre um número: "))
    contador = contador + 1
    total = total + numero
media = total / contador
print("A média é " + str(media))
```

Exemplo:

```
contador = 0
while contador < 9:
   print('O contador é:', contador)
   contador = contador + 1
print("Goodbye!")
```

# Comando de Repetição (*For*)

Executa uma sequência de comandos múltiplas vezes e abrevia o código que gerencia a variável de laço.

Exemplo:

```
for x in range(0, 5):
    print("Índice %d" % (x))
```

Exemplo:

```
for x in range(1, 11):
    for y in range(1, 11):
        print("%d * %d = %d" % (x, y, x*y))
```

# Comando de Repetição (*For*)

Exemplo:

```
for x in range(3):
    print(x)
    if x == 1:
        break
```

Exemplo:

```
str = "Hello, World!"
for x in str:
    print(x)
```

Exemplo:

```
frutas = ['morango', 'banana', 'melancia', 'manga']
for i in range(len(frutas)):
    print('Fruta corrente :', frutas[i])
print("Goodbye!")
```

# Funções

Maneira conveniente de dividir um código em porções menores, de forma a proporcionar melhor modularidade, manutenibilidade, reusabilidade e interpretabilidade.

Exemplo:

```
def imprime_resultado(nota):
  if nota >= 50:
     print('Aprovado')
  else:
     print('Reprovado')
  return

imprime_resultado(51)
```

Saída:

```
Aprovado
```

Exemplo:

```
def resultado(nota):
  if nota >= 50:
     return 'Aprovado'
  else:
     return 'Reprovado'

resultado(92)
```

Saída:

```
'Aprovado'
```

# Passagem por Referência versus Valor

Todos os parâmetros (argumentos) na linguagem Python são passados por referência, o que significa que se a referência de um parâmetro for alterada no interior de uma função, a mudança também refletirá fora da função.

Exemplo:

```
# definição da função
def altera_lista(lista):
   lista.append([1, 2, 3, 4]);
   print("Valores dentro da função: ", lista)
   return

# chamada da função
lista = [10, 20, 30];
altera_lista(lista);
print("Valores fora da função: ", lista)
```

```
Valores dentro da função:  [10, 20, 30, [1, 2, 3, 4]]
Valores fora da função:  [10, 20, 30, [1, 2, 3, 4]]
```

# Escopo de Variáveis

- O escopo de uma variável determina a parte do programa em que se pode acessar um identificador específico. Há dois escopos básicos de variáveis em Python: variáveis locais e variáveis globais.
- Variáveis que são definidas dentro do corpo de uma função têm um escopo local, e aquelas definidas fora têm um escopo global. Isso significa que variáveis locais podem ser acessadas apenas dentro da função na qual foram declaradas, enquanto variáveis globais podem ser acessadas por todo o corpo do programa por todas as funções.

```
total = 0  # variável global
# definição da função
def soma(arg1, arg2):
   total = arg1 + arg2  # total é uma variável local aqui
   print("Total: ", total)
   return total;

# chamada da função
soma(10, 20);
print("Total: ", total)
```

Saída:

```
Total:  30
Total:  0
```

# Módulos

- Um módulo permite que a organização lógica de um código Python. Agrupar código relacionado em um módulo torna o código mais fácil de entender e usar.
- Um módulo é um objeto Python com atributos arbitrariamente nomeados aos quais se pode vincular e referenciar.
- Basicamente, um módulo é um arquivo contendo código Python que pode definir funções, classes e variáveis.

Exemplo: module funcoes.py

```
def cubo(y):
    return y * y * y

def dobro(z):
    return 2 * z
```

# Comando import

Um arquivo de origem Python pode ser usado como um módulo executando uma declaração de importação em algum outro arquivo de origem Python.

```
import module1, module2, ..., moduleN
```

Exemplo:

```
import funcoes

for x in range(1, 6):
    print(funcoes.cubo(x), end=" ")
print()
print()

for x in range(1, 6):
    print(funcoes.dobro(x), end=" ")
```

Saída:

```
1 8 27 64 125

2 4 6 8 10
```

# Comando `from...import`

A instrução `from` da linguagem Python permite que se importe atributos específicos de um módulo para o espaço de nomes atual.

```
from modname import name1, name2, ..., nameN
```

Exemplo: importa a função para a sequência de Fibonacci do módulo `fib`:

```
from fib import fibonacci
```

# Comando `from...import *`

Também é possível importar todos os nomes de um módulo para o espaço de nomes atual.

```
from modname import *
```

Embora isso forneça uma maneira fácil de importar todos os itens de um módulo, essa declaração deve ser usada com moderação.

# Entrada de Dados via Teclado

A função `input()` é utilizada para receber entrada do usuário via teclado.

Exemplo:

```
str = input("Digite entrada: ");
print("Entrada é: ", str)
```

Saída (quando "Hello, World!" é digitada):

```
Digite entrada: Hello, World!
Entrada é: Hello, World!
```

Exemplo:

```
str = input("Digite entrada: ");
str = eval(str)
print("Entrada é: ", str)
```

Saída:

```
Digite entrada: [x*5 for x in range(2, 10, 2)]
Entrada é: [10, 20, 30, 40]
```

# Saída de Dados na Tela

- A função `print()` permite que zero ou mais expressões separadas por vírgulas sejam passadas como argumentos.
- A função converte as expressões passadas em uma cadeia de caracteres e escreve o resultado na saída padrão.

Exemplo:

```
print("Exemplo de cadeia de caracteres")

a = 10
v = 3.14
s = "Hello World!"
print(a, v, s)
```

Saída:

```
Exemplo de cadeia de caracteres
10 3.14 Hello World!
```

# Abertura e Fechamento de Arquivos

O Python fornece funções e métodos básicos necessários para manipular arquivos por padrão. A maioria das manipulações de arquivo pode ser realizada usando um objeto de arquivo.

`open` `file` `close` `read` `write` `tell` `seek` `rename` `remove`

Exemplo:

```
# abre um arquivo
f = open("entrada.txt", "r+")
str = f.read(10);
print("Sequência lida: ", str)

# verifica posição corrente
posicao = f.tell();
print("Posição corrente do arquivo: ", posicao)

# reposiciona ponteiro no início do arquivo
position = f.seek(0, 0);
str = f.read(10);
print("Sequência lida novamente: ", str)
# fecha arquivo
f.close()
```

Saída (assumindo que o arquivo de entrada tenha a sequência "Python language"):

```
Sequência lida:  Python lan
Posição corrente do arquivo:  10
Sequência lida novamente:  Python lan
```

# Vetores NumPy

Um vetor armazena e representa dados regulares de forma estruturada.

Exemplo:

```
import numpy as np
a = np.zeros(shape=(3, 2))
print(a)
```

Saída:

```
array([[ 0. 0.],
       [ 0. 0.],
       [ 0. 0.]])
```

Exemplo:

```
a[0] = [1, 2]
a[1] = [2, 3]
print(a)
```

Saída:

```
array([[ 1. 2.],
       [ 2. 3.],
       [ 0. 0.]])
```

# Vetores NumPy

Exemplos:

```
# cria uma matriz de 1s
np.ones((3, 4))

# verifica a forma de uma matriz
x = np.ones((3, 4))
print(x.shape)

# cria uma matriz de 0s
np.zeros((2, 3, 4), dtype=np.int16)

# cria uma matriz com valores aleatórios
np.random.random((2,2))

# cria uma matriz vazia
np.empty((3, 2))

# cria uma matriz com uma constante a ser inserida na matriz
np.full((2, 2), 7)

# cria uma matriz de valores espaçados uniformemente
np.arange(10, 25, 5)

# cria uma matriz de valores espaçados uniformemente
np.linspace(0, 2, 9)
```

# Vetores NumPy

Equivalente aos operadores `+`, `-`, `*`, `/` or `%`, as funções `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()` e `np.remainder()` podem ser usadas para realizar operações aritméticas com vetores.

Algumas outras funções úteis:

```
a.sum()           # soma dos elementos do vetor
a.min()           # valor mínimo do vetor
a.max(axis=0)     # valor máximo ao longo das colunas do vetor
a.cumsum(axis=1) # soma cumulativa dos elementos ao longo das linhas do vetor
a.cumprod()       # produto cumulativo dos elementos começando de 1
a.mean()          # média
a.median()        # mediana
a.corrcoef()      # coeficiente de correlação
np.var()          # variância
np.std(a)         # desvio padrão
```

# Vetores NumPy

- Os eixos são definidos para vetores com mais de uma dimensão.
- Em um vetor 2D, pode-se aplicar uma operação verticalmente ao longo das colunas (eixo 0) ou horizontalmente ao longo das linhas (eixo 1).

```
x = np.arange(12).reshape((3, 4))
print(x)
print(x.sum(axis=0))
print(x.sum(axis=1))
```

```
[[ 0  1  2  3],
 [ 4  5  6  7],
 [ 8  9 10 11]]

[12 15 18 21]
[ 6 22 38]
```

# Vetores NumPy

- Diferenças entre multiplicação de matrizes e multiplicação de elementos de matrizes:

Exemplo:

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# produto de matrizes
x = np.dot(a,b)
print(x)

# multiplicação de matrizes elemento a elemento (produto de Hadamard)
y = np.multiply(a,b)
print(y)

# multiplicação de matrizes elemento a elemento (produto de Hadamard)
z = a * b
print(z)
```

Saída:

```
[[19 22],
 [43 50]]

[[ 5 12],
 [21 32]]

[[ 5 12],
 [21 32]]
```

# Vetores NumPy

Operações para acesso aos elementos de um vetor 1D.

```
vetor1d = np.arange(10)
print(vetor1d)
```

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

```
print(vetor1d[5])
```

```
5
```

```
print(vetor1d[5:8])
```

```
[5 6 7]
```

```
vetor1d[5:8] = 12
print(vetor1d)
```

```
[ 0  1  2  3  4 12 12 12  8  9]
```

# Vetores NumPy

Operações para acesso aos elementos de um vetor 2D.

```
vetor2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(vetor2d[2])
```

```
[7 8 9]
```

```
print(vetor2d[0][2])
```

```
3
```

```
print(vetor2d[:2])
```

```
[[1 2 3],
 [4 5 6]]
```

```
print(vetor2d[0:2, 1])
```

```
[2 5]
```

```
print(vetor2d[:2, 1:])
```

```
[[2 3],
 [5 6]]
```

```
print(vetor2d[1:3, 2])
```

```
[6 9]
```

```
print(vetor2d[1:5:2, ::3])
```

```
[[ 4]
 [10]]
```

# Vetores NumPy

Operações para acesso aos elementos de um vetor 3D.

```
vetor3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(vetor3d)
```

```
[[[ 1  2  3],
  [ 4  5  6]],
 [[ 7  8  9],
  [10 11 12]]]
```

```
print(vetor3d[0])
```

```
[[1 2 3],
 [4 5 6]]
```

```
print(vetor3d[1, 0])
```

```
[7 8 9]
```

```
print(vetor3d[1,...])
```

```
[[ 7  8  9]
 [10 11 12]]
```

# Vetores NumPy

Em resumo:

```
a[start:end]  # itens do início até o final (mas o final não está incluído)
a[start:]     # itens do início até o final do vetor
a[:end]       # itens do começo até o final (mas o final não está incluído)
```

# Vetores NumPy

```
# seleciona valores de um vetor que atendam a uma determinada condição
a = np.random.random((1, 7))
print(a[a < 0.6])
```

Duas maneiras de gerar a transposta de um vetor bidimensional:

```
print(np.transpose(a))
print(a.T)
```

# Vetores NumPy

- Os vetores podem ser remodelados ou redimensionados para torná-los compatíveis com as operações aritméticas desejadas.
- Para redimensionar um vetor, suas novas dimensões podem ser passadas para a função `np.resize()`. Se o novo vetor for maior do que o original, ele será preenchido com cópias do vetor original que são repetidas quantas vezes forem necessárias.

Exemplo:

```
a = np.array([[0, 1], [2, 3]])
np.resize(a, (2, 3))
```

```
array([[0, 1, 2],
       [3, 0, 1]])
```

```
np.resize(a, (1, 4))
```

```
array([[0, 1, 2, 3]])
```

```
np.resize(a, (2, 4))
```

```
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
```

# Vetores NumPy

Também é possível alterar a forma de um vetor sem alterar seus dados.

```
x = np.ones((3, 4))

# imprime o tamanho de x
print(x.size)

# reformata x para (2, 6)
y = x.reshape((2, 6))
```

Outra operação para alterar a forma de vetores é `ravel()`, que é usada para converter um vetor $n$-dimensional em um vetor 1D.

```
# converte x para vetor 1D
z = x.ravel()

# imprime z
print(z)
```

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

# Vetores NumPy

É possível anexar vetores a um vetor original, onde eles são adicionados ao final desse vetor original.

```
# anexa um vetor 1D para vetor 'vetor1d'
vetor1d = np.arange(7)
novo_vetor1d = np.append(vetor1d, [7, 8, 9, 10])

# imprime novo vetor
print(vetor1d)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10]
```

```
# anexa uma coluna extra para vetor 'vetor2d'
vetor2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
novo_vetor2d = np.append(vetor2d, [[-1], [-2] , [-3], [-4]], axis=1)

# imprime novo vetor
print(novo_vetor2d)
```

```
[[ 1  2  3 -1]
 [ 4  5  6 -2]
 [ 7  8  9 -3]
 [10 11 12 -4]]
```

# Vetores NumPy

Há funções para inserir e remover elementos.

```
# insere '5' no índice 1
np.insert(vetor1d, 1, 5)

# remove o valor no índice 1
np.delete(vetor1d, [1])
```

# Vetores NumPy

Há várias funções para unir ou concatenar vetores.

```
# concatena 'vetor1d' e 'x'
print(np.concatenate((vetor1d, x)))

# empilha vetores linha por linha
print(np.vstack((vetor1d, vetor2d)))

# empilha vetores linha por linha
print(np.r_[novo_vetor1d, vetor2d])

# empilha vetores horizontalmente
print(np.hstack((novo_vetor1d, vetor2d)))

# empilha vetores coluna por coluna
print(np.column_stack((novo_vetor1d, vetor2d)))

# empilha vetores coluna por coluna
print(np.c_[novo_vetor1d, vetor2d])

# divide 'vetor_empilhado' horizontalmente no segundo índice
print(np.hsplit(vetor_empilhado, 2))

# divide 'vetor_empilhado' verticalmente no segundo índice
print(np.vsplit(vetor_empilhado, 2))
```

# Computação Vetorizada

A computação vetorizada de vetores fornece uma maneira rápida de calcular operações para manipular dados, como filtragem, transformação, sumarização e agregação.

Exemplo: operações matemáticas realizadas em blocos inteiros de dados usando uma sintaxe semelhante às operações equivalentes entre elementos escalares.

```
A = np.array([[0.84, -0.26,  0.92],
              [0.42,  0.31, -0.73]])

B = A * 10
print(B)

C = A + A
print(C)
```

```
[[ 8.4 -2.6  9.2]
 [ 4.2  3.1 -7.3]]

[[ 1.68 -0.52  1.84]
 [ 0.84  0.62 -1.46]]
```

Para definir todos os valores negativos no vetor como 0:

```
A[A < 0] = 0
```

```
[[ 0.84 0.   0.92]
 [ 0.42 0.31 0.  ]]
```

# Computação Vetorizada

Exemplo: produto interno $Y = X^T X$

```
X = np.random.randn(6, 3)
Y = np.dot(X.T, X)
print(Y)
```

```
[[ 4.27527809 -0.24339761 -2.65985356]
 [-0.24339761  4.76032665 -1.07444714]
 [-2.65985356 -1.07444714  5.46999821]]
```

Para vetores com altas dimensões, o comando de transposição aceita uma tupla de números de eixos para permutar os eixos:

```
X = np.arange(16).reshape((2, 2, 4))
print(X)
Y = X.transpose((1, 0, 2))
print(Y)
```

```
[[[ 0,  1,  2,  3],
  [ 4,  5,  6,  7]],
 [[ 8,  9, 10, 11],
  [12, 13, 14, 15]]]

[[[ 0,  1,  2,  3],
  [ 8,  9, 10, 11]],
 [[ 4,  5,  6,  7],
  [12, 13, 14, 15]]]
```

# Computação Vetorizada

A instrução `numpy.where` é uma versão vetorizada da expressão ternária `x if condition else y`.

Exemplo: Um valor de `vetor_x` deve ser considerado sempre que a condição `cond` for verdadeira; caso contrário, o valor de `vetor_y` correspondente deve considerado. Uma solução possível é:

```
vetor_x = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
vetor_y = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])

resultado = [(x if c else y)
                for x, y, c in zip(vetor_x, vetor_y, cond)]
print(resultado)
```

```
[1.1, 2.2, 1.3, 1.4, 2.5]
```

Esta solução tem alguns problemas, não sendo rápida para vetores grandes. Além disso, não funcionará com vetores multidimensionais. Uma solução mais concisa com `numpy.where` é:

# Computação Vetorizada

A instrução `numpy.where` é uma versão vetorizada da expressão ternária `x if condition else y`.

Exemplo: Um valor de `vetor_x` deve ser considerado sempre que a condição `cond` for verdadeira; caso contrário, o valor de `vetor_y` correspondente deve considerado. Uma solução possível é:

```
vetor_x = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
vetor_y = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])

resultado = [(x if c else y)
                for x, y, c in zip(vetor_x, vetor_y, cond)]
print(resultado)
```

```
[1.1, 2.2, 1.3, 1.4, 2.5]
```

Esta solução tem alguns problemas, não sendo rápida para vetores grandes. Além disso, não funcionará com vetores multidimensionais. Uma solução mais concisa com `numpy.where` é:

```
resultado = np.where(cond, vetor_x, vetor_y)
print(resultado)
```

```
[ 1.1, 2.2, 1.3, 1.4, 2.5]
```

# Computação Vetorizada

A função `where` pode ser usada para expressar lógicas mais complicadas. Considere dois vetores booleanos, `cond1` e `cond2`, em que se deseja atribuir um valor diferente para cada um dos 4 pares possíveis de valores booleanos:

```
resultado = []
for i in range(n):
    if cond1[i] and cond2[i]:
        resultado.append(0)
    elif cond1[i]:
        resultado.append(1)
    elif cond2[i]:
        resultado.append(2)
    else:
        resultado.append(3)
```

O laço `for` pode ser convertido em uma expressão `where` aninhada:

# Computação Vetorizada

A função `where` pode ser usada para expressar lógicas mais complicadas. Considere dois vetores booleanos, `cond1` e `cond2`, em que se deseja atribuir um valor diferente para cada um dos 4 pares possíveis de valores booleanos:

```
resultado = []
for i in range(n):
    if cond1[i] and cond2[i]:
        resultado.append(0)
    elif cond1[i]:
        resultado.append(1)
    elif cond2[i]:
        resultado.append(2)
    else:
        resultado.append(3)
```

O laço `for` pode ser convertido em uma expressão `where` aninhada:

```
np.where (cond1 & cond2, 0,
     np.where (cond1, 1,
          np.where (cond2, 2, 3)))
```

# Computação Vetorizada

Funções matemáticas e estatísticas podem ser calculadas em todo um vetor ou ao longo de um eixo.

Exemplos:

```
x = np.random.randn(5, 4)  # dados normalmente distribuídos

print(x.mean())
print(np.mean(x))
print(x.sum())
print(x.mean(axis=1))
print(x.sum(0))
```

```
array([[ 1.63724392,  0.82355426, -0.14623522,  1.20527739],
       [ 0.33014103,  0.3454366 , -0.58564072,  0.84797426],
       [ 0.51293932, -0.18159329, -0.10766887,  0.30108391],
       [ 0.75550274,  0.51283271,  1.39118016, -0.62332175],
       [ 0.33780799, -1.05125264,  1.08284995,  0.53570564]])

0.3961908693845762 6
0.3961908693845762
7.923817387691525
[0.87996009 0.23447779 0.13119027 0.50904846 0.22627774]
[3.573635   0.44897765 1.6344853  2.26671945]
```

# Computação Vetorizada

Vetores NumPy podem ser ordenados usando o método `sort`:

```
x = np.random.randn(6)
print(x)

x.sort()
print(x)
```

```
[-2.0282963   0.13228264 -0.19936979 -0.06106798 -0.13287158 -1.33947695]

[-2.0282963  -1.33947695 -0.19936979 -0.13287158 -0.06106798  0.13228264]
```

# Computação Vetorizada

- NumPy possui algumas operações básicas de conjunto para vetores unidimensionais.
- Uma operação comum é `np.unique`, que retorna os valores únicos ordenados em um vetor:

```
nomes = np.array(['Carlos', 'Pedro', 'Maria', 'Carlos',
                  'Maria', 'Pedro', 'Pedro', 'Marcos'])
print(np.unique(nomes))

inteiros = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
print(np.unique(inteiros))
```

```
['Carlos' 'Marcos' 'Maria' 'Pedro']
[1 2 3 4]
```

# Computação Vetorizada

A função `np.in1d` testa a associação dos valores de um vetor em outro, retornando um vetor booleano:

```
valores = np.array([6, 0, 0, 3, 2, 5, 6])
print(np.in1d(valores, [2, 3, 6]))
```

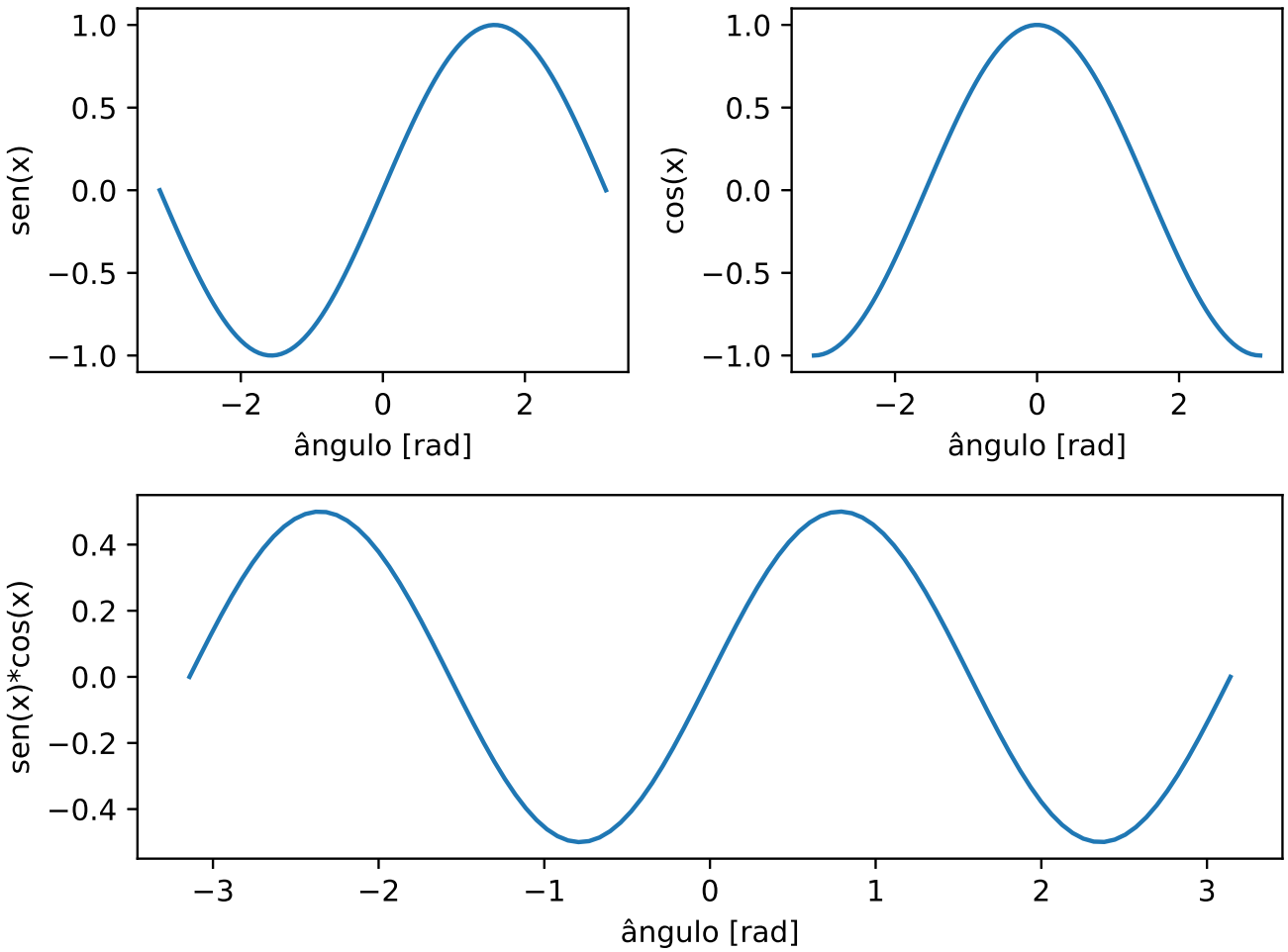
```
[ True False False  True  True False  True]
```

# Computação Vetorizada

Para calcular e exibir os valores do seno e cosseno de um vetor de ângulos dados em radianos:

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

x = np.linspace(-np.pi, np.pi, 100)

plt.subplot(2,2,1)
plt.plot(x, np.sin(x))
plt.xlabel('ângulo [rad]')
plt.ylabel('sen(x)')

plt.subplot(2,2,2)
plt.plot(x, np.cos(x))
plt.xlabel('ângulo [rad]')
plt.ylabel('cos(x)')
plt.axis('tight')

plt.subplot(2,1,2)
plt.plot(x, np.sin(x) * np.cos(x))
plt.xlabel('ângulo [rad]')
plt.ylabel('sen(x)*cos(x)')
plt.tight_layout()
plt.show()
```

# Computação Vetorizada



# Imagens com NumPy and SciPy

Realiza alguns cálculos e processamentos em uma imagem.

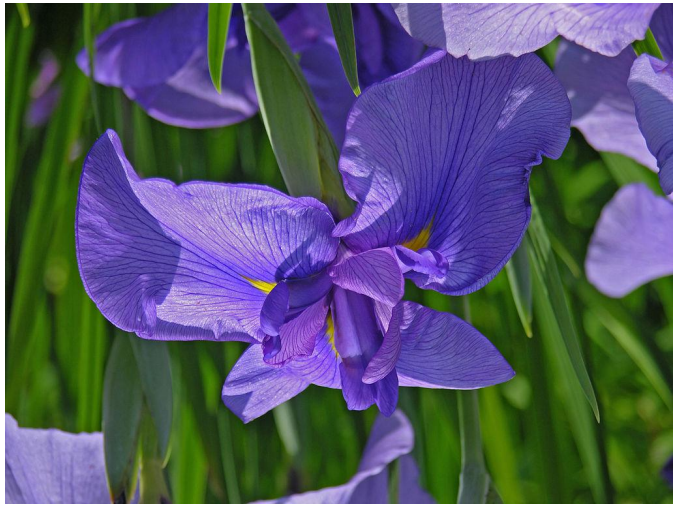
```
from scipy import misc
from scipy import ndimage
import numpy as np
import matplotlib.pyplot as plt

# abre imagem e armazena-a em um vetor
img = misc.imread('iris.png')

# imprime dimensões e tipo da imagem
print(img.shape, img.dtype)

# mostra imagem
plt.imshow(img, cmap='gray')
plt.show()

# salva imagem em formato PNG
misc.imsave('iris.png', img)

# calcula algumas informações estatísticas
print(img.min(), img.mean(), img.max())
```

# Imagens com NumPy and SciPy

```
# aplica transformação de rotação
f = np.flipud(img)
plt.imshow(f)
plt.show()

# suaviza imagem com filtro gaussiano
g = ndimage.gaussian_filter(img, sigma=7)
h = ndimage.gaussian_filter(img, sigma=11)
plt.imshow(g)
plt.show()
plt.imshow(h)
plt.show()
```

# Computação Vetorizada

Saída:

# Exercícios

- Inicializar uma matriz $M_{5\times10}$ com valor 7.

Versão não vetorizada:

# Exercícios

- Inicializar uma matriz $M_{5\times10}$ com valor 7.

Versão não vetorizada:

```
M = np.empty([5, 10])
for i in range(5):
    for j in range(10):
        M[i][j] = 7
```

Versão vetorizada 1:

# Exercícios

- Inicializar uma matriz $M_{5\times10}$ com valor 7.

Versão não vetorizada:

```
M = np.empty([5, 10])
for i in range(5):
    for j in range(10):
        M[i][j] = 7
```

Versão vetorizada 1:

```
M = np.full((5, 10), 7)
```

Versão vetorizada 2:

# Exercícios

- Inicializar uma matriz $M_{5\times10}$ com valor 7.

Versão não vetorizada:

```
M = np.empty([5, 10])
for i in range(5):
    for j in range(10):
        M[i][j] = 7
```

Versão vetorizada 1:

```
M = np.full((5, 10), 7)
```

Versão vetorizada 2:

```
M = np.ones((5, 10)) * 7
```

# Exercícios

- Vetorizar o seguinte código:

```
for i in range(200):
    for j in range(400):
        if dados[i][j] > 0:
           dados[i][j] = 0
```

Versão vetorizada:

# Exercícios

- Vetorizar o seguinte código:

```
for i in range(200):
    for j in range(400):
        if dados[i][j] > 0:
           dados[i][j] = 0
```

Versão vetorizada:

```
dados[dados > 0] = 0
```

# Exercícios

- Dada a matriz a seguir, imprimir (i) o número de elementos maiores que 8 e (ii) a soma dos elementos maiores que 8.

```
M = np.array([[13, 24, 1, 8, 15],
              [23, 5, 7, 14, 16],
              [4, 6, 13, 20, 22],
              [10, 12, 19, 21, 3],
              [11, 18, 25, 2, 9]])
```

Versão vetorizada:

# Exercícios

- Dada a matriz a seguir, imprimir (i) o número de elementos maiores que 8 e (ii) a soma dos elementos maiores que 8.

```
M = np.array([[13, 24, 1, 8, 15],
              [23, 5, 7, 14, 16],
              [4, 6, 13, 20, 22],
              [10, 12, 19, 21, 3],
              [11, 18, 25, 2, 9]])
```

Versão vetorizada:

```
# número de elementos maiores que 8
print(np.sum(M > 8))

# soma dos elementos maiores que 8
print(np.sum(M[M > 8]))
```

```
17
285
```

# Exercícios

- Reverter um vetor.

Versão não vetorizada:

# Exercícios

- Reverter um vetor.

Versão não vetorizada:

```
a = np.arange(10)
b = np.copy(a)
n = len(a)
for i in range(n):
    b[i] = a[n-i-1]
```

Versão vetorizada 1:

# Exercícios

- Reverter um vetor.

Versão não vetorizada:

```
a = np.arange(10)
b = np.copy(a)
n = len(a)
for i in range(n):
    b[i] = a[n-i-1]
```

Versão vetorizada 1:

```
b = a[::-1]
```

Versão vetorizada 2:

# Exercícios

- Reverter um vetor.

Versão não vetorizada:

```
a = np.arange(10)
b = np.copy(a)
n = len(a)
for i in range(n):
    b[i] = a[n-i-1]
```

Versão vetorizada 1:

```
b = a[::-1]
```

Versão vetorizada 2:

```
b = np.flipud(a)
```

# Exercícios

- Obter a média de cada coluna de um vetor 2D, ignorando elementos menores ou iguais a 0.

Versão vetorizada 1:

# Exercícios

- Obter a média de cada coluna de um vetor 2D, ignorando elementos menores ou iguais a 0.

Versão vetorizada 1:

```
a = np.array([[2.0, 4.0, 5.0], [-3.0, 8.0, -7.0], [0.0, 6.0, 4.0]])
masked_a = np.ma.masked_array(a, mask=(a <= 0))
media = np.ma.mean(masked_a, axis=0)
print(media)
```

```
[2.  6.  4.5]
```

Versão vetorizada 2:

# Exercícios

- Obter a média de cada coluna de um vetor 2D, ignorando elementos menores ou iguais a 0.

Versão vetorizada 1:

```
a = np.array([[2.0, 4.0, 5.0], [-3.0, 8.0, -7.0], [0.0, 6.0, 4.0]])
masked_a = np.ma.masked_array(a, mask=(a <= 0))
media = np.ma.mean(masked_a, axis=0)
print(media)
```

```
[2.  6.  4.5]
```

Versão vetorizada 2:

```
a = np.array([[2.0, 4.0, 5.0], [-3.0, 8.0, -7.0], [0.0, 6.0, 4.0]])
mascara = a > 0
col_soma = np.sum(a * mascara, axis=0)
contadores = np.sum(mascara, axis=0)
media = np.where(contadores > 0, col_soma / contadores, np.nan)
print(media)
```

```
[2.  6.  4.5]
```

# Exercícios

- Remover de um vetor todos os elementos que são iguais ao maior elemento.

# Exercícios

- Remover de um vetor todos os elementos que são iguais ao maior elemento.

```
x = np.array([6, 0, 0, 3, 2, 5, 6])
valor_max = np.max(x)
indices = np.where(x == valor_max)
novo_x = np.delete(x, indices)
print(novo_x)
```

```
array([0, 0, 3, 2, 5])
```

# Exercícios

- Um vetor `x` contém alguns 0s. Criar `y` tal que `y[i]` será 0 se `x[i]` for 0, caso contrário, `y[i]` será `log(x[i])`.

# Exercícios

- Um vetor `x` contém alguns 0s. Criar `y` tal que `y[i]` será 0 se `x[i]` for 0, caso contrário, `y[i]` será `log(x[i])`.

```
x = np.array([0, 2, 3, 5, 0, 7, 9, 0, 4])
indices = x > 0
y = np.zeros(len(x))
y[indices] = np.log(x[indices])
```

# Exercícios

- Calcular a soma dos elementos ao longo da diagonal principal de uma matriz quadrada `M`.

Versão não vetorizada:

# Exercícios

- Calcular a soma dos elementos ao longo da diagonal principal de uma matriz quadrada `M`.

Versão não vetorizada:

```
M = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])
contador = 0.0
for i in range(0, len(M)):
    contador += M[i][i]
print(contador)
```

Versão vetorizada 1:

# Exercícios

- Calcular a soma dos elementos ao longo da diagonal principal de uma matriz quadrada `M`.

Versão não vetorizada:

```
M = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])
contador = 0.0
for i in range(0, len(M)):
    contador += M[i][i]
print(contador)
```

Versão vetorizada 1:

```
indices = np.arange(M.shape[0])
diagonal = M[indices, indices]
print(np.sum(diagonal))
```

Versão vetorizada 2:

# Exercícios

- Calcular a soma dos elementos ao longo da diagonal principal de uma matriz quadrada `M`.

Versão não vetorizada:

```
M = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])
contador = 0.0
for i in range(0, len(M)):
    contador += M[i][i]
print(contador)
```

Versão vetorizada 1:

```
indices = np.arange(M.shape[0])
diagonal = M[indices, indices]
print(np.sum(diagonal))
```

Versão vetorizada 2:

```
print(M.trace())
```

# Exercícios

- Verificar se uma matriz `M` is triangular inferior, triangular superior ou diagonal.

# Exercícios

- Verificar se uma matriz `M` is triangular inferior, triangular superior ou diagonal.

```
# verifica se é uma matriz triangular inferior
np.allclose(M, np.tril(M))

# verificar se é uma matriz triangular superior
np.allclose(M, np.triu(M))

# verifica se uma matriz é diagonal
np.allclose(M, np.diag(np.diag(M)))
```

# Exercícios

- Vetorizar o seguinte código:

```
def limiar(M, max_value, min_value):
    n = M.shape[0]
    m = M.shape[1]
    for i in range(n):
        for j in range(m):
            c = M[i][j]
            if c > valor_max:
               M[i][j] = valor_max
            elif c < valor_min:
               M[i][j] = valor_min
    return M
```

Versão vetorizada:

# Exercícios

- Vetorizar o seguinte código:

```
def limiar(M, max_value, min_value):
    n = M.shape[0]
    m = M.shape[1]
    for i in range(n):
        for j in range(m):
            c = M[i][j]
            if c > valor_max:
               M[i][j] = valor_max
            elif c < valor_min:
               M[i][j] = valor_min
    return M
```

Versão vetorizada:

```
def limiar(M, valor_max, valor_min):
    M[M > valor_max] = valor_max
    M[M < valor_min] = valor_min
    return M
```

# Exercícios

- Vetorizar o seguinte código:

```
x = np.linspace(-2., 2., 5)
y = np.zeros(x.shape)

for i in range(len(x)):
    if x[i] <= 0.5:
        y[i] = x[i]**2
    else:
        y[i] = -x[i]
```

Versão vetorizada:

# Exercícios

- Vetorizar o seguinte código:

```
x = np.linspace(-2., 2., 5)
y = np.zeros(x.shape)

for i in range(len(x)):
    if x[i] <= 0.5:
        y[i] = x[i]**2
    else:
        y[i] = -x[i]
```

Versão vetorizada:

```
y = np.where(x <= 0.5, x**2, -x)
```

# Exercícios

- Computação de $seno(x)$.

Versão não vetorizada:

```
n = 1000000
x = np.linspace(0, 1, n+1)

def seno(x):
    res = np.zeros_like(x)
    for i in range(len(x)):
        res[i] = np.sin(x[i])
    return res

y = seno(x)
```

Versão vetorizada:

# Exercícios

- Computação de $seno(x)$.

Versão não vetorizada:

```
n = 1000000
x = np.linspace(0, 1, n+1)

def seno(x):
    res = np.zeros_like(x)
    for i in range(len(x)):
        res[i] = np.sin(x[i])
    return res

y = seno(x)
```

Versão vetorizada:

```
y = np.sin(x)
```

# Exercícios

- Soma duas listas de inteiros.

Uma maneira simples usando Python puro é:

```
def soma_python(Z1,Z2):
    return [z1+z2 for (z1,z2) in zip(Z1,Z2)]
```

Esta primeira solução pode ser vetorizada usando NumPy:

```
def soma_numpy(Z1,Z2):
    return np.add(Z1,Z2)
```

Não apenas o segundo método é mais rápido, mas também se adapta naturalmente à forma de Z1 e Z2. Esta é a razão pela qual não se deve escrever Z1 + Z2, pois não funcionaria se Z1 e Z2 forem ambas listas.

# Exercícios

No primeiro método em Python puro, o operador + é interpretado de forma diferente dependendo da natureza dos dois objetos, de modo que se duas listas aninhadas forem consideradas, obtém-se as seguintes saídas:

```
Z1 = [[1, 2], [3, 4]]
Z2 = [[5, 6], [7, 8]]
Z1 + Z2
```

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

```
soma_python(Z1, Z2)
```

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

```
soma_numpy(Z1, Z2)
```

```
[[ 6  8]
 [10 12]]
```

O primeiro método concatena as duas listas, o segundo método concatena as listas internas e o último calcula o que é (numericamente) esperado.

# Exercícios

- Dados dois vetores `X` e `Y`, deseja-se calcular a soma de `X[i]*Y[j]` para todos os pares de índices `i`, `j`. Uma solução simples e óbvia é escrever:

```
def calcula_python(X, Y):
    resultado = 0
    for i in range(len(X)):
        for j in range(len(Y)):
            resultado += X[i] * Y[j]
    return resultado
```

No entanto, esta primeira implementação requer dois laços e é lenta. Uma maneira de vetorizar o problema é identificar a partir de conceitos de álgebra linear que a expressão `X[i]*Y[j]` é muito semelhante a uma expressão de produto de matriz. Portanto, pode-se obter algum ganho de velocidade com NumPy. Uma solução incorreta é:

```
def calcula_numpy_errado(X, Y):
    return (X*Y).sum()
```

Ela está errada porque a expressão `X*Y` na verdade calculará um novo vetor `Z` tal que `Z[i] = X[i]*Y[i]` e isso não é o que se quer. Em vez disso, pode-se explorar o mecanismo de *broadcasting* do NumPy, primeiro remodelando os dois vetores e depois multiplicando-os:

```
def calcula_numpy(X, Y):
    Z = X.reshape(len(X),1) * Y.reshape(1,len(Y))
    return Z.sum()
```

# Exercícios

Aqui tem-se que `Z[i,j] == X[i,0]*Y[0,j]` e se a soma for considerada sobre cada elemento de `Z`, obtém-se o resultado esperado, o que é superior por um fator de aproximadamente 150 (assumindo um vetor de 1000 elementos).

Pode-se fazer ainda melhor. Se a versão em Python puro for observada em detalhes, pode-se notar que o laço interno não depende do índice `i`.

```
def calcula_numpy_melhor_1(X, Y):
    resultado = 0
    for i in range(len(X)):
        Ysum = 0
        for j in range(len(Y)):
            Ysum += Y[j]
        resultado += X[i] * Ysum
    return resultado
```

Isto significa que o laço interno pode ser removido do laço externo.

```
def calcula_numpy_melhor_2(X, Y):
    resultado = 0
    Ysum = 0
    for j in range(len(Y)):
        Ysum += Y[j]
    for i in range(len(X)):
        resultado += X[i] * Ysum
    return resultado
```

# Exercícios

A remoção do laço interno transforma a complexidade de $O(n^2)$ em $O(n)$. Usando a mesma abordagem, pode-se escrever:

```
def calcula_numpy_melhor_3(x, y):
    Ysum = 0
    for j in range(len(y)):
        Ysum += y[j]
    Xsum = 0
    for i in range(len(y)):
        Xsum += y[i]
    return Xsum * Ysum
```

Finalmente, percebendo que apenas o produto da soma sobre `X` e `Y`, respectivamente, é necessário, pode-se aproveitar a função `np.sum` e escrever:

```
def calcula_numpy_melhor(x, y):
    return np.sum(y) * np.sum(x)
```

Este código é mais compacto, mais claro e muito mais rápido.

O problema foi reformulado, aproveitando o fato de que $\sum_{i,j} X_i Y_j = \sum_i X_i \sum_j Y_j$.

# Exercícios

- Em resumo, há dois tipos principais de vetorização: vetorização de código e vetorização de problema.
- A segunda abordagem é a mais difícil, entretanto, onde se pode esperar ganhos maiores de velocidade.
- No último problema, ganhou-se um fator de 150 com a vetorização de código, mas um fator de 70.000 com a vetorização de problema (assumindo um vetor de 1000 elementos), apenas escrevendo o problema de forma diferente (embora não se possa esperar uma melhoria tão grande em todas as situações).
- No entanto, a vetorização de código continua sendo um fator importante e se a última solução no estilo Python for reescrita, o ganho é interessante, mas não tanto quanto na versão NumPy:

```
def calcula_python_melhor(x, y):
    return sum(x) * sum(y)
```

- Esta nova versão em Python puro é muito mais rápida do que a versão anterior, mas ainda é 50 vezes mais lenta do que a versão em NumPy.

# Exercícios

- Contar o número de transições de `False` para `True` em uma sequência.

```
Exemplo: x = np.array([False, True, False, False, True])
```

```
Saída: 2
```

- Selecionar apenas os números ímpares e elevá-los ao quadrado.

```
Exemplo: x = np.array([1, 2, 3, 4])
```

```
Saída: [1, 9]
```

- Somar todos os elementos de um vetor que são divisíveis por 5.

```
Exemplo: x = np.array([1, 5, 12, 15, 20, 22])
```

```
Saída: 40
```

- Somar subsequências de $n$ valores em um vetor.

```
Exemplo: x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]) e n = 4
```

```
Saída: [10, 26, 42]
```

- Trocar todos os valores negativos de um vetor por zeros.

```
Exemplo: x = np.array([-3, -2, -1, 0, 1, 2, 3])
```

```
Saída: [0, 0, 0, 0, 1, 2, 3]
```

# Exercícios

- Contar o número de valores pares em um vetor.

```
Exemplo: x = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
Saída: 3
```

- Encontrar os máximos (picos) locais em um vetor.

```
Exemplo: x = np.array([1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 2,
                       1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1])
```

```
Saída: [5, 3, 6]
```

- Calcular a média móvel com uma janela de tamanho $n$.

```
Exemplo: x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) e n = 4
```

```
Saída: [2.5, 3.5, 4.5, 5.5, 6.5, 7.5]
```

- Normalizar um vetor para que seus elementos somem 1.

```
Exemplo: x = np.array([1, 2, 3, 4])
```

```
Saída: [0.1, 0.2, 0.3, 0.4]
```

- Calcular a distância euclidiana entre dois vetores.

```
Exemplo:
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
```

```
Saída: 5.19615
```

# Exercícios

- Verificar se um vetor está ordenado crescentemente (retorno: `True` ou `False`).

```
Exemplo: v = np.array([1, 2, 3, 4, 5])
```

```
Saída: True
```

- Calcular as diferenças consecutivas de um vetor.

```
Exemplo: v = np.array([10, 7, 4, 3, 2])
```

```
Saída: [-3, -3, -1, -1]
```

- Somar os elementos positivos de um vetor.

```
Exemplo: v = np.array([1, -2, 3, 4, -5, 6, -7, 8, -9])
```

```
Saída: 22
```

- Verificar se um vetor é um palíndromo.

```
Exemplo: x = np.array([1, 2, 3, 4, 5, 4, 3, 2, 1])
```

```
Saída: True
```

- Construir o histograma de uma imagem monocromática.