MAP3122: Métodos numéricos e aplicações

Quadrimestral 2021

Antoine Laurain

Introdução a Python

Introdução

Porque usar Python?

- Python é open-source.
- Python tem uma sintaxe simples e é fácil de aprender e usar em comparação com outras linguagens como C++.
- Python possui muitas bibliotecas (desenvolvimento web, cálculo científico, configuração de nuvem, análise de dados, aprendizado de máquina, IA e pesquisa acadêmica), o que o torna uma ferramenta muito versátil.
- Python teve um grande crescimento nos últimos 10 anos e se tornou uma das principais linguagens de programação em universidades e indústrias.
- O conhecimento de Python oferece muitas oportunidades de carreira.

Porque usar NumPy?

- NumPy significa Numerical Python.
- NumPy é uma biblioteca Python usada para trabalhar com ferramentas de álgebra linear, matrizes e vetores.
- Em Python podemos trabalhar com listas mas o processamento delas é lento. Para calculos numéricos de álgebra linear precisamos usar matrizes e vetores de NumPy que são muito mais rápidos (até 50 vezes mais rápido).

Informações práticas para usar Python

- É altamente recomendável usar `Python3` em vez de `Python2`. Um código em `Python3` pode ser rodado desta maneira num terminal:

  ```
  python3 mycode.py
  ```

- Para ver se `Python3` esta instalado no seu computador, pode digitar no terminal:

  ```
  python3 --version
  ```

A indentação em Python

- A indentação (recuo) refere-se aos espaços no início de uma linha de código.
- Enquanto em outras linguagens de programação a indentação no código é apenas para legibilidade, a indentação em Python é **obrigatória**.
- Python usa indentação para indicar um bloco de código.
- Python foi desenvolvido para ser uma linguagem de fácil leitura, com um visual agradável, frequentemente usando palavras e não pontuações como em outras linguagens. A indentação obrigatória em Python é uma característica importante que facilita a leitura.
- Exemplo de indentação:

```
if 5 > 2:
    print("Five is greater than two!")
```

- Exemplo de falta de indentação que gera um erro em Python:

```
if 5 > 2:
print("Five is greater than two!")
```

- O número de espaços na indentação é com você, mas deve ser pelo menos um. **Porém, é recomando usar** 4 **espaços para a indentação**, pois isso é o mais usado na communidade de Python.
- Um exemplo com duas indentações:

```
for i in range(0,2):
    for j in range(0,2):
        print("something")
```

Importar bibliotecas

- Nas primeiras linhas do código importamos bibliotecas:

```
from numpy import *
```

  Isso quer dizer que importamos todas as funções da biblioteca NumPy, por exemplo:

```
from numpy import *
v = array([1, 2, 3, 4, 5])
```

- Podemos importar também funções específicas da biblioteca:

```
from numpy import array, concatenate
v1 = array([1, 2, 3])
v2 = array([4, 5, 6])
v = concatenate((v1, v2))
```

- Em geral é melhor importar uma biblioteca usando uma abreviação para evitar colisões de nomes entre bibliotecas:

```
import numpy as np
```

  Isso quer dizer que toda função de NumPy deverá ser chamada na forma `np.function`, por exemplo:

```
import numpy as np
v = np.array([1, 2, 3, 4, 5])
```

Usando Python interativamente

- Para fazer debugging do seu código, precisamos de ferramentas interativas. Recomando em particular duas ferramentas:
- O **iPython** permite trabalhar com Python de maneira interativa: `https://pypi.org/project/ipython/`
- A biblioteca **pdb** (pdb = Python Debugger) permite fazer debugging de códigos usando pontos de interrupção.
- A principal aplicação da biblioteca `pdb` é o uso da função `set_trace()`:

```
import numpy as np
from pdb import set_trace

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

  A função `set_trace()` cria um ponto de interrupção no programa que permite fazer debugging a partir deste ponto.
- Cuidado: a função `set_trace()` tem os próprios comandos tais como `n,p,s,c,u,` ... que têm prioridade sobre as variaveis do seu código. Para accessar sua variavel com o mesmo nome, tem que escrever por exemplo `!n,!p,!s`, etc ...

Primeiros passos com NumPy

- Definir um vetor:

```
import numpy as np

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

print(v)
```

- Verificar a versão de NumPy:

```
import numpy as np

print(np.__version__)
```

- Criar uma matriz (2D-array):

```
import numpy as np

M = np.array([[1, 2, 3], [4, 5, 6]])

print(M)
```

Primeiros passos com NumPy

- Obtenha o primeiro elemento do vetor v:

```
import numpy as np

v = np.array([1, 2, 3, 4])

print(v[0])
```

- Acesse o elemento na posição $[0, 1]$ da matriz M (primeira linha, segunda coluna):

```
import numpy as np

M = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st dim: ', M[0, 1])
```

Primeiros passos com NumPy

“Fatiar” vetores:

- “Fatia” os elementos do índice 1 ao índice 5 da seguinte matriz:

```
import numpy as np

v = np.array([1, 2, 3, 4, 5, 6, 7])

print(v[1:5])
```

- Retorne todos os elementos do índice 1 ao índice 5 com passo de 2:

```
import numpy as np

v = np.array([1, 2, 3, 4, 5, 6, 7])

print(v[1:5:2])
```

- Funciona também para matrizes:

```
import numpy as np

M = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(M[1, 1:4])
```

Algumas funções úteis de NumPy

- A função `reshape` permite transformar vetores em matrizes ou matrizes em vetores:

```
import numpy as np

v = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
P = v.reshape(4, 3)
print("P = ",P)

M = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
u = M.reshape(10)
print("u = ",u)
```

- A função `np.multiply(A,B)` é uma multiplicação elementos por elementos das matrizes A e B, enquanto `np.dot(A,B)` é o produto matricial das matrizes A e B (o resultado é diferente!). Podemos usar também a notação mais simples `A*B = np.multiply(A,B)`:

```
import numpy as np

A = np.array([[1, 2, 3], [1, 4, 5], [6, 7, 8]])
B = np.array([[1, 1, 0], [0, 1, 1], [0, 1, 0]])

print("A = ",A)
print("B = ",B)

print("A*B = ",A*B)
print("np.multiply = ",np.multiply(A,B))
print("np.dot(A,B) = ",np.dot(A,B))
```

Funções

- Funções

```
import numpy as np

def f(x,y):
    a = 2.0
    return x**2 + a*x + y

print(f(2.0,1.0))
```

- Exemplo com matrizes

```
import numpy as np

def f(n):
    a = 2.3
    return a + np.eye(n)

print(f(6))
```

Algumas funções úteis de NumPy

- O produto de Kronecker de $A \in \mathbb{R}^{m_1 \times n_1}$ e $B \in \mathbb{R}^{m_2 \times n_2}$ é uma matriz em bloco $A \otimes B \in \mathbb{R}^{m_1 m_2 \times n_1 n_2}$ definida por

$$A \otimes B = \begin{pmatrix} AB_{11} & AB_{12} & \dots & AB_{1n_2} \\ AB_{21} & AB_{22} & \dots & AB_{2n_2} \\ \vdots & \vdots & \ddots & \ddots \\ AB_{m_2 1} & AB_{m_2 2} & \dots & AB_{m_2 n_2} \end{pmatrix}.$$

  Aqui, cada bloco AB_{ij} é uma matriz de tamanho $m_1 \times n_1$. O produto de Kronecker em NumPy é dado por:

- Exemplo com $m_1 = n_1 = m_2 = n_2 = 2$:

$$B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \qquad A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

  então

$$A \otimes B = \texttt{numpy.kron(B,A)} = \begin{bmatrix} AB_{11} & AB_{12} \\ AB_{21} & AB_{22} \end{bmatrix} = \begin{bmatrix} A & 2A \\ 3A & 4A \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 0 & 2 & 0 \\ 3 & 3 & 4 & 4 \\ 3 & 0 & 4 & 0 \end{bmatrix}$$

```
import numpy as np
A = np.array([[1,1], [1,0]])
B = np.array([[1,2], [3,4]])
M = np.kron(B,A)
print(M)
```

Exercícios

- Monta a matriz M seguinte usando a função `numpy.kron`:

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Exercícios

- Monta a matriz M seguinte usando a função `numpy.kron`:

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

- Solução: o primeiro passo é de identificar o bloco:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
import numpy as np

A = np.eye(2)
B = np.array([[1,1], [1,0]])
M = np.kron(B,A)
print(M)
```

Exercícios

- Monta a matriz M seguinte usando a função `numpy.kron`:

$$M = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 3 \\ 2 & 2 & 0 & 0 \\ 2 & 2 & 0 & 0 \end{bmatrix}$$

  A dificuldade agora é que precisamos usar dois blocos diferentes:

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

Exercícios

- Monta a matriz M seguinte usando a função `numpy.kron`:

$$M = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 3 \\ 2 & 2 & 0 & 0 \\ 2 & 2 & 0 & 0 \end{bmatrix}$$

  A dificuldade agora é que precisamos usar dois blocos diferentes:

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

- Solução:

$$M = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 2 & 2 & 0 & 0 \end{bmatrix}$$

```
import numpy as np

A1 = np.eye(2)
B1 = np.array([[1,3], [0,0]])
A2 = np.ones((2,2))
B2 = np.array([[0,0], [2,0]])

M = np.kron(B1,A1) + np.kron(B2,A2)
print(M)
```

Exercícios

- Dados n, m e uma matriz $A \in \mathbb{R}^{n \times n}$, escreve uma função que constroi a matriz $M \in \mathbb{R}^{nm \times nm}$ seguinte usando a função `numpy.kron` (isso quer dizer que têm m blocos A horizontalmente e verticalmente na matriz M):

$$M = \begin{bmatrix} A & \cdots & A \\ \vdots & \ddots & \vdots \\ A & \cdots & A \end{bmatrix}$$

Exercícios

- Dados n, m e uma matriz $A \in \mathbb{R}^{n \times n}$, escreve uma função que constroi a matriz $M \in \mathbb{R}^{nm \times nm}$ seguinte usando a função `numpy.kron` (isso quer dizer que têm m blocos A horizontalmente e verticalmente na matriz M):

$$M = \begin{bmatrix} A & \cdots & A \\ \vdots & \ddots & \vdots \\ A & \cdots & A \end{bmatrix}$$

- Solução:

```
import numpy as np
m = 5
n = 2
A = np.array([[1,3], [7,0]])
B = np.ones((m,m))
M = np.kron(B,A)
print(M)
```