Universidade Federal Rural do Semi-Árido
Centro Multidisciplinar de Pau dos Ferros
Ciências Exatas e Naturais

Curso de Introdução a machine learning com Python 3 (Professores: Rosana Cibely Batista Rego (Coordenadora), Bruno Fontes de Sousa, Ferdinandy Silva Chagas, Samara Martins Nascimento)

Aula de Introdução ao Numpy

Prof.Bruno Fontes de Sousa

28 de fevereiro de 2023

Sumário I

Sumário II

Sumário III

Sumário IV

Seção 1. Introdução ao numpy

O que é e para que serve o Numpy?

- **O que é Numpy?** Numpy é uma biblioteca de cálculo numérico para a linguagem de programação Python. Ela fornece uma forma eficiente de manipular grandes conjuntos de dados numéricos.
- **Por que usar o Numpy?** Numpy é mais rápido e mais fácil de usar que outras bibliotecas de Python para manipulação de dados. Ele fornece recursos para realizar cálculos matemáticos avançados de forma eficiente.
- **Características do Numpy**
  - O Numpy usa elementos do tipo Array N-dimensional.
  - O Numpy tem suporte para álgebra linear, cálculos matemáticos avançados (como a transformada de Fourier) e geração de números aleatórios.
  - O Numpy tem pode ser integrado com outras bibliotecas de ciência de dados, como pandas e Matplotlib.

Instalação e importação do Numpy

A biblioteca Numpy pode ser instalada das seguintes maneiras.

- Usando o gerenciador de pacotes **pip**, com os comandos `pip install numpy`.
- Com o **Anaconda**, usando o **Anaconda Navigator** ou em um terminal usando o código `conda install numpy`.
- Ou no **Jupyter Notebook** com a linha de comando `!pip install numpy`.

Para usar o Numpy dentro de um script do Python, precisamos importá-lo. Geralmente, importamos com a seguinte linha de comando:
`import numpy as np`
Em seguida, podemos começar a criar e manipular arrays Numpy.

Seção 2. Formatos de números no numpy

Formatos de números no numpy

Formato de número inteiro:

- np.int0
- np.int8
- np.int16
- np.int32
- np.int64

Formato float:

- np.float16
- np.float32
- np.float64
- np.float128

Formato de número complexo:

- np.complex64
- np.complex128
- np.complex256
- np.complex_
- np.singlecomplex
- np.longcomplex

Formatos de números no numpy

Há diversos outros formatos para trabalhar com dados no numpy. A lista de todos os formatos pode ser obtida com o comando

```
lst = list(set(np.sctypeDict.values()))
lst = [str(i) for i in lst]
lst = sorted(lst, key=lambda x:x[14])
for value in lst:
    print(value)
```

Formatos de números no numpy

A saída será a seguinte:

```
<class 'numpy.bytes_'>
<class 'numpy.bool_'>
<class 'numpy.complex128'>
<class 'numpy.complex64'>
<class 'numpy.complex256'>
<class 'numpy.datetime64'>
<class 'numpy.float128'>
<class 'numpy.float16'>
<class 'numpy.float64'>
<class 'numpy.float32'>
<class 'numpy.int32'>
<class 'numpy.int64'>
<class 'numpy.int8'>
<class 'numpy.int16'>
<class 'numpy.longlong'>
<class 'numpy.object_'>
<class 'numpy.str_'>
<class 'numpy.timedelta64'>
<class 'numpy.uint64'>
<class 'numpy.ulonglong'>
<class 'numpy.uint8'>
<class 'numpy.uint16'>
<class 'numpy.uint32'>
<class 'numpy.void'>
```

Formatos de números no numpy

Para mais informações sobre os formatos de dados do numpy, copie o nome da função, por exemplo, `numpy.timedelta64`, e coloque no Google ou outro buscador da sua escolha. Vocês podem buscar da seguinte forma:

numpy.timedelta64 site:numpy.org

e assim a busca será feita especificamente no no site oficial da bibliioteca NumPy.

Convertendo de um formato para outro

Quando convertemos um número inteiro para o formato float, não há nenhum problema, porque o formato float serve para representar qualquer número real (inteiro ou decimal).
Exemplo:

```
#Convertendo de inteiro para float
print(np.float64(2)) # Saída: 2.0
```

Mas, quando convertemos do formato float para o formato de número inteiro, o resultado só será o mesmo se o número que se quer converter já é um número inteiro. Se um número for um decimal, o resultado da conversão será a sua parte inteira.
Exemplo:

```
#Convertendo de float para inteiro
print(np.int64(2.0)) # Saída: 2
print(np.int64(2.1)) # Saída: 2
```

Seção 3. Arrays

Array

- **O que é um array?** Um array é uma estrutura de dados que permite armazenar vários dados (números, vetores, matrizes, etc) em uma única variável.
- **Por que usar arrays?**
  - Maior eficiência em operações matemáticas em comparação com listas.
  - Capacidade de realizar operações em todos os elementos do array de uma só vez.
- **Como criar arrays no Numpy?** Usando a função `numpy.array()`.
  Exemplo:
  ```
  import numpy
  a = numpy.array([1, 2, 3])
  ou
  import numpy as np
  a = np.array([1, 2, 3])
  ```

Observação

Tipos de dados de um array

Quando você usa `numpy.array` para definir um novo array, você deve considerar o tipo (dtype) dos elementos que compõem o array, que pode ser especificado explicitamente. Se você não tiver cuidado com as atribuições do dtype, você pode obter um resultado errado para o array informado.

Exemplo:

```
import numpy as np
arr1 = np.array([127, 128, 129], dtype=np.int8)
print(arr1) #Saída: [127 -128 -127]
print(arr1.dtype) #Saída: int8
arr2 = np.array([127, 128, 129], dtype=np.int16)
print(arr2) #Saída: [127 128 129]
print(arr2.dtype) #Saída: int16
arr3 = np.array([127, 128, 129])
print(arr3) #Saída: [127 128 129]
print(arr3.dtype) #Saída: int64
```

Dimensão e forma de um array

Para determinar a **dimensão** de um array usamos o atributo `ndim` e para determinar a **forma** de um array, usamos o atributo `shape`.
Exemplo: Array 1D

```
import numpy as np
arr = np.array([1, 2, 3])
print("Dimensão do array:", arr.ndim)
#Dimensão do array: 1
print("Forma do array:", arr.shape)
#Forma do array: (3,)
```

Dimensão e forma de um array

Exemplo: Array 2D

```
#arr = np.array([[1, 2, 3], [4, 5, 6]])
arr = np.array([
    [1, 2, 3],
    [4, 5, 6]
])
print("Dimensão do array:", arr.ndim)
# Dimensão do array: 2
print("Forma do array:", arr.shape)
#Forma do array: (2, 3)
```

Dimensão e forma de um array

Exemplo: Array 3D

```
import numpy as np
arr = np.array([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]],
    [[13, 14, 15], [16, 17, 18]],
    [[19, 20, 21], [22, 23, 24]],
    ])
print("Dimensão do array:", arr.ndim)
# Dimensão do array: 3
print("Forma do array:", arr.shape)
#Forma do array: (4, 2, 3)
```

Exercício para fazer em casa: Array 4D

Tente obter a forma de cada um dos seguintes arrays. Depois de tentar, verifique se acertou com os comandos `arr1.shape` e `arr2.shape`.

```
import numpy as np

arr1 = np.array([
    [[[9, 5], [7, 7], [8, 3]], [[9, 4], [1, 5], [7, 0]]],
    [[[5, 8], [0, 3], [0, 8]], [[4, 9], [7, 2], [4, 5]]]
])

arr2 = np.array([
    [[[3,6],[3,6],[8,1]], [[9,9],[3,5],[9,4]], [[7,5],[4,0],[3,7]], [[1,2],[6,5],[0,3]]],
    [[[4,8],[5,0],[3,3]], [[9,1],[9,9],[2,8]], [[6,3],[1,0],[9,6]], [[4,1],[7,7],[6,7]]],
    [[[6,2],[3,9],[9,6]], [[2,4],[4,7],[3,3]], [[9,0],[0,3],[4,1]], [[9,1],[2,1],[0,1]]]
])
```

Funções np.reshape e np.resize

Introdução

- As funções np.reshape() e np.resize() são usadas para redimensionar arrays.
- A principal diferença entre as duas funções é que np.reshape() retorna um novo array com o mesmo tamanho dos dados de entrada, enquanto np.resize() pode redimensionar o array para um tamanho maior ou menor.

Função np.reshape

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.reshape():

```
import numpy as np
a = np.array([1, 2, 3, 4, 5, 6])
b = np.reshape(a, (2, 3))
print(b) # Saída: [[1 2 3]
# [4 5 6]]
```

- Nesse exemplo, a função np.reshape() é usada para redimensionar o array unidimensional "a"para um array bidimensional com 2 linhas e 3 colunas.
- O novo array é atribuído à variável "b"e impresso na tela.

Função np.resize

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.resize():

```
import numpy as np
a = np.array([1, 2, 3])
b = np.resize(a, (2, 4))
print(b) # Saída: [[1 2 3 1]
# [2 3 1 2]]
```

- Nesse exemplo, a função np.resize() é usada para redimensionar o array unidimensional "a"para um array bidimensional com 2 linhas e 4 colunas.
- O novo array é atribuído à variável "b"e impresso na tela.
- Observe que a função np.resize() preenche o novo array com os elementos do array de entrada repetidamente, se necessário.

Função np.transpose

Introdução

- A função np.transpose() é usada para transpor um array.
- A transposição de um array significa que as linhas e colunas do array são trocadas entre si.
- Em outras palavras, as linhas se tornam colunas e as colunas se tornam linhas.

Função np.transpose

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.transpose():

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.transpose(a)
print(b) # Saída: [[1 3]
# [2 4]]
```

- Nesse exemplo, a função np.transpose() é usada para transpor o array "a".
- O array transposto é atribuído à variável "b"e impresso na tela.

Função np.ravel

Introdução

- A função np.ravel() é usada para retornar um array unidimensional que contém os elementos do array original na mesma ordem.
- O array retornado é uma visão do array original, ou seja, alterar o array retornado também altera o array original.
- Essa função é semelhante a np.flatten(), mas o array retornado por np.ravel() é uma visão, enquanto o array retornado por np.flatten() é uma cópia.

Função np.ravel

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.ravel():

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.ravel(a)
print(b) # Saída: [1 2 3 4]
```

- Nesse exemplo, a função np.ravel() é usada para retornar um array unidimensional que contém os elementos do array "a"na mesma ordem.
- O array retornado é atribuído à variável "b"e impresso na tela.

Método flatten

Introdução

- O método flatten é usado para criar uma cópia unidimensional de um array multidimensional.
- A cópia é criada de tal forma que os elementos do array original são colocados na nova dimensão na mesma ordem em que aparecem no array original.
- Ao contrário da função np.ravel(), o método flatten retorna sempre uma cópia (independente) do array original, mesmo que o array original seja unidimensional.

Método flatten

Exemplo de uso

- O exemplo a seguir ilustra o uso do método flatten:

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = a.flatten()
print(b) # Saída: [1 2 3 4]
```

- Nesse exemplo, a função np.flatten() é usada para criar uma cópia unidimensional do array "a".
- O array retornado é atribuído à variável "b"e impresso na tela.

Diferença entre a função np.ravel e o método flatten

Introdução

- A funçção np.ravel() e o método flatten são usados para transformar um array multi-dimensional em um array unidimensional.
- Ambos retornam um array com os mesmos elementos do array original, mas com a forma unidimensional.
- A principal diferença entre os dois é que np.ravel() retorna um array que fica vinculado ao original, enquanto que o método flatten retorna uma cópia independente do array original.

Diferença entre a função np.ravel e o método flatten

Exemplo de uso

O exemplo a seguir ilustra a diferença entre a função np.ravel() e o método flatten:

```
import numpy as np
a1 = np.array([[1, 2], [3, 4]])
b1 = np.ravel(a1) #Planificando o array a1 com a função np.ravel.
b1[0] = 0 #Alterando a primeira entrada de b1
print(a1) # Saída: np.array([[0, 2], [3, 4]]) #a1 também foi alterado
print(b1)  # Saída: np.array([[1, 2], [3, 4]])
```

```
import numpy as np
a2 = np.array([[1, 2], [3, 4]]) #Observe que a1 e a2 são iguais.
b2 = a2.flatten() #Planificando o array a2 com o método np.ndarray.flatten()
b2[0] = 0 #Alterando a primeira entrada do array b2
print(a2) # Saída: np.array([[1, 2], [3, 4]]) #a2 não foi alterado
print(b2)  # Saída: np.array([[0, 2], [3, 4]])
```

Função np.squeeze

Introdução

- A função np.squeeze() é usada para remover dimensões de tamanho 1 de um array.
- Isso é útil, por exemplo, quando queremos converter um array multidimensional em um array unidimensional, ou quando queremos remover dimensões redundantes de um array.
- A função retorna um novo array sem as dimensões de tamanho 1. O array original não é modificado.

Função np.squeeze

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.squeeze():

```
import numpy as np
a = np.array([[[1], [2]], [[3], [4]]])
b = np.squeeze(a)
print(b) # Saída: [[1 2]
# [3 4]]
```

- Nesse exemplo, o array "a"tem dimensões (2, 2, 1), ou seja, 2 planos com 2 linhas e 1 coluna cada.
- A função np.squeeze() é usada para remover a dimensão de tamanho 1 do array "a".
- O novo array é atribuído à variável "b"e impresso na tela.

Função np.expand_dims

Introdução

- A função np.expand_dims() é usada para adicionar uma nova dimensão com tamanho 1 em um array.
- Essa função pode ser útil para realizar operações com arrays que exigem que os arrays tenham o mesmo número de dimensões.
- A posição em que a nova dimensão deve ser adicionada é especificada pelo argumento "axis".

Função np.expand_dims

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.expand_dims():

```
import numpy as np
a = np.array([1, 2, 3])
b = np.expand_dims(a, axis=0)
print(b) # Saída: [[1 2 3]]
```

- Nesse exemplo, a função np.expand_dims() é usada para adicionar uma nova dimensão com tamanho 1 ao array unidimensional "a".
- A nova dimensão é adicionada na posição 0 (primeira posição) do array.
- O array resultante é atribuído à variável "b"e impresso na tela.

Adição de Arrays

Para calcular a **adição** de dois arrays, basta usar o operador "+".
Exemplo:

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
c = a + b
print(c)
```

Resultado:

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

Subtração de Arrays

A **subtração** de arrays funciona de maneira análoga que a adição, trocando o operador "+" pelo operador "-".

Exemplo:

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
c = a - b
print(c)
```

Resultado:

```
[-4 -4 -4 -4]
```

Multiplicação de Arrays

A **multiplicação** de arrays no NumPy é realizada elemento a elemento, assim como os operadores "+" e "-".
Exemplo:

```
import numpy as np
array_1 = np.array([1, 2, 3, 4])
array_2 = np.array([10, 20, 30, 40])
produto = array_1 * array_2
print(produto)
```

A saída será:

```
[10 40 90 160]
```

Divisão de Arrays

A **divisão** de arrays no NumPy também é realizada elemento a elemento.
Exemplo:

```
import numpy as np
array_1 = np.array([10, 20, 30, 40])
array_2 = np.array([1, 2, 3, 4])
divisao = array_1 / array_2
print(divisao)
```

A saída será:

```
[10. 10. 10. 10.]
```

Operações de arrays com números

Também é possível operar um array com um número fixo. Será feita a operação do número com cada elemento do array, gerando um novo array.

Exemplo:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr+3) #Saída: [4 5 6 7]
print(arr-3) #Saída: [-2 -1  0  1]
print(arr*3) #Saída: [ 3  6  9 12]
print(arr/3) #Saída: [0.33333333 0.66666667 1. 1.33333333]
```

Operações de arrays com números

Exemplo:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(3+arr) #Saída: [4 5 6 7]
print(3-arr) #Saída: [ 2  1  0 -1]
print(3*arr) #Saída: [ 3  6  9 12]
print(3/arr) #Saída: [3.   1.5  1.   0.75]
```

Função np.logical_and

- A função np.logical_and() é usada para calcular a operação lógica "and"(e) entre dois arrays de valores booleanos.
- A função retorna um array com os valores booleanos resultantes da operação "and"elemento por elemento entre os dois arrays.

```
#Exemplo de uso da função np.logical_and()
import numpy as np
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
c = np.logical_and(a, b)
print(c) # Saída: [ True False False False]
```

Função np.logical_or

Introdução

- A função np.logical_or realiza uma operação booleana "or"elemento por elemento em dois arrays booleanos.
- Ela retorna um novo array booleano com o resultado da operação booleana.
- A operação booleana "or"é definida como:

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Função np.logical_or

Exemplo

```
#Exemplo de uso da função np.logical_or
import numpy as np
a = np.array([True, False, True, False])
b = np.array([False, True, True, False])
c = np.logical_or(a, b)
print(c) # Saída: [ True True True False]
```

Nesse exemplo, a função np.logical_or é usada para realizar a operação booleana "or"elemento por elemento nos arrays "a"e "b". O resultado é um novo array booleano "c"com o resultado da operação. Os elementos de "c"são [True, True, True, False], que são o resultado da operação booleana "or"em cada elemento de "a"e "b".

Função np.logical_xor

Introdução

- A função np.logical_xor é usada para calcular o ou-exclusivo (XOR) entre dois arrays booleanos.
- O resultado da operação entre dois elementos booleanos é True somente se ambos forem diferentes.
- A função np.logical_xor retorna um array booleano com o resultado da operação elemento por elemento entre os dois arrays.

Função np.logical_xor

Exemplo

```
#Exemplo de uso da função np.logical_xor
import numpy as np
a = np.array([True, False, True, False])
b = np.array([False, True, True, False])
c = np.logical_xor(a, b)
print(c) # Saída: [ True True False False]
```

Nesse exemplo, a função np.logical_xor é usada para calcular o ou-exclusivo elemento por elemento entre os arrays "a"e "b". O resultado da operação é um array booleano com o valor [True, True, False, False], que representa o resultado da operação entre os elementos correspondentes dos arrays "a"e "b".

Função np.logical_not

Introdução

- A função np.logical_not() é usada para aplicar a negação lógica (not) elemento por elemento em um array booleano.
- A função retorna um novo array com o resultado da negação lógica em cada elemento do array de entrada.
- O array de entrada não é modificado.

Função np.logical_not

Exemplo de uso

- O exemplo a seguir ilustra o uso da função np.logical_not():

```
import numpy as np
a = np.array([True, False, True])
b = np.logical_not(a)
print(b) # Saída: [False True False]
```

- Nesse exemplo, a função np.logical_not() é usada para aplicar a negação lógica elemento por elemento no array booleano "a".
- A função retorna um novo array booleano com o resultado da negação lógica em cada elemento do array de entrada.
- O novo array é atribuído à variável "b"e impresso na tela.

Função np.any

Introdução

- A função np.any() do NumPy é usada para verificar se pelo menos um elemento de um array é True.
- Ela retorna True se pelo menos um elemento do array é True e False caso contrário.
- A função pode ser usada para verificar se um array contém pelo menos um valor válido (por exemplo, não NaN ou não infinito).

Função np.any

Exemplo de uso

- A função np.any() pode ser usada da seguinte forma:

```
import numpy as np
a = np.array([False, False, True, False])
if np.any(a):
    print("Pelo menos um elemento é True.")
else:
    print("Todos os elementos são False.")
```

- Nesse exemplo, a função np.any() é usada para verificar se pelo menos um elemento do array "a"é True.
- Como o terceiro elemento é True, a mensagem "Pelo menos um elemento é True."é exibida.

Função np.all

Introdução

- A função np.all() é uma função universal do NumPy que retorna True se todos os elementos de um array forem avaliados como True.
- Ela pode ser usada em arrays booleanos e em arrays com outros tipos de dados.
- É uma função útil para verificar se todos os elementos de um array são True, por exemplo.

Função np.all

Exemplos de uso

- Exemplo de uso da função np.all() em um array booleano:

```
import numpy as np
a = np.array([True, True, False, True])
if np.all(a):
    print("Todos os elementos de a são True.")
else:
    print("Pelo menos um elemento de a é False.")
```

- Saída: "Pelo menos um elemento de a é False."

Função np.all

Exemplos de uso

- Exemplo de uso da função np.all() em um array com outros tipos de dados:

```
import numpy as np
a = np.array([1, 2, 3, 4])
if np.all(a > 0):
    print("Todos os elementos de a são positivos.")
else:
    print("Pelo menos um elemento de a é negativo.")
```

- Saída: "Todos os elementos de a são positivos."

Seção 4. Indexação e fatiamento de arrays

Indexação de arrays

Indexando com valores positivos

O primeiro elemento de um array 1-dimensional tem índice igual a zero, o segundo tem índice igual a um, e assim sucessivamente.
O último elemento do array tem índice igual ao "tamanho do array menos 1", ou seja, arr.size-1 = len(arr)-1.

```
import numpy as np
arr = np.array([3, 7, 2, 5, 1, 3, 2, 2, 5, 1])
print("1º elemento:", arr[0]) #Saída: 1º elemento: 3
print("2º elemento:", arr[1]) #Saída: 2º elemento: 7
print("3º elemento:", arr[2]) #Saída: 3º elemento: 2
#...
print("8º elemento:", arr[7]) #Saída: 8º elemento: 2
print("9º elemento:", arr[8]) #Saída: 9º elemento: 5
print("10º elemento:", arr[9]) #Saída: 10º elemento: 1
```

Indexação de arrays

Indexando com valores negativos

Também é possível identificar os elementos de um array 1-dimensional usando índices negativos. Neste caso, o índice "-1" seleciona o último elemento, o índice "-2" seleciona o penúltimo, o índice "-3" seleciona o antepenúltimo, e assim por diante. Nesse caso, o primeiro elemento terá índice igual a "-len(arr)".

```
import numpy as np
arr = np.array([3, 7, 2, 5, 1, 3, 2, 2, 5, 1])
print(arr[-1]) #Saída: 1
print(arr[-2]) #Saída: 5
print(arr[-3]) #Saída: 2
#...
print(arr[-8]) #Saída: 2
print(arr[-9]) #Saída: 7
print(arr[-10]) #Saída: 3
```

Fatiamento (slicing) de arrays

Fatiar um array **unidimensional** significa recortar "pedaços" do array. Ou seja, obter novos arrays que contém elementos que vão de um índice até outro índice do array original. Este processo segue a mesma lógica de fatiamento de outros tipos de sequências do python, como listas e tuplas.
Também é possível fatiar um array **multidimensional**.

Fatiamento (slicing) de arrays unidimensionais

Considerando `i<j`, temos que

- `arr[i:j]` seleciona todos os elementos que tem índice k tal que $i \leq k \leq j-1$.
- `arr[:j]` seleciona todos os elementos que tem índice k tal que $0 \leq k \leq j-1$.
- `arr[::-1]` resulta em um novo array que é igual ao array original na ordem reversa.
- `arr[i:j:2]` seleciona elementos que tem índice k tal que $i \leq k \leq j-1$ e "saltando" sempre um elemento. Ou seja, começa do arr[i], pula o elemento arr[i+1], inclui o elemento arr[i+2], pula o elemento arr[i+3] e assim por diante.
- `arr[i:j:3]` seleciona elementos que tem índice k tal que $i \leq k \leq j-1$ e "saltando" dois elementos em cada etapa.
- `arr[j:]` seleciona os elementos da posição `j` para frente. Se `j>0`, então `arr[-j:]` seleciona os `j` últimos elementos do array.
- Se `j>0`, então `arr[:-j]` remove os `j` últimos elementos do array.

Fatiamento (slicing) de arrays unidimensionais

Exemplos

```
import numpy as np
arr = np.arange(10) #Saída: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

print(arr[2:6]) #Saída: [2 3 4 5]
print(arr[:6]) #Saída: [0 1 2 3 4 5]
print(arr[::-1]) #Saída: [9 8 7 6 5 4 3 2 1 0]
print(arr[2:6:2]) #Saída: [2 4]
print(arr[2:6:3]) #Saída: [2 5]
print(arr[6:]) #Saída: [6 7 8 9]
print(arr[:-3]) #Saída: [0 1 2 3 4 5 6]
```

Fatiamento de arrays bidimensionais

Considerando um array **bidimensional** de forma (shape) igual a (m, n).

- `arr[i,j]` seleciona o elemento da linha `i` e coluna `j` do array.
- `arr[i:j,k]` seleciona todos os elementos da linha i até a linha $j-1$ na coluna k.
- `arr[i]` ou `arr[i,:]` seleciona toda a linha i.
- `arr[:,k]` seleciona toda a coluna k.
- `arr[i,k:m]` seleciona elementos da linha i que vão da coluna k até a coluna $m-1$.
- `arr[i:j,k:m]` seleciona um array (uma submatriz) que vai da linha i até a linha $j-1$ e da coluna k até a coluna $m-1$.

Fatiamento de arrays bidimensionais

Exemplos

```
import numpy as np
arr = np.array([[2, 5, 8, 0, 2, 7],
                [1, 4, 3, 9, 6, 7],
                [9, 7, 4, 0, 1, 4],
                [3, 8, 9, 1, 2, 5]])
print(arr[1,4]) #Saída: 6
print(arr[1:3,4]) #Saída: [6 1]
print(arr[1,:]) #Saída: [1 4 3 9 6 7]
print(arr[:,4]) #Saída: [2 6 1 2]
print(arr[1,2:5]) #Saída: [3 9 6]
print(arr[1:3,2:5])
"""Saída: [[3 9 6]
          [4 0 1]]"""
```

Fatiamento de arrays multidimensionais

Considerando agora um array **multidimensional**. O símbolo de reticências, $\cdots$, significa o mesmo que uma sequência de : (dois pontos). Por exemplo, se arr é um array 5-dimensional, então:

- $\mathrm{arr}[1, 2, \cdots]$ é o mesmo que $\mathrm{arr}[1, 2, :, :, :]$.
- $\mathrm{arr}[\cdots, 3]$ é o mesmo que $\mathrm{arr}[:, :, :, :, 3]$.
- $\mathrm{arr}[4, \cdots, 5, :]$ é o mesmo que $\mathrm{arr}[4, :, :, :, 5, :]$.

(NUMPY..., s.d., p. 13)

Seção 5. Aplicação de Funções aos Elementos de um Array

Funções universais do numpy

Numpy possui funções universais que podem ser aplicadas elemento por elemento de um array retornando um novo array com os resultados.

- Exponenciais e logarítmicas: np.exp(), np.exp2(), np.log(), np.log10() e np.log2().
- Funções trigonométricas: np.sin, np.cos, np.tan, np.arcsin, np.arccos, np.arctan, np.deg2rad e np.rad2deg.
- Funções hiperbólicas: np.sinh, np.cosh, np.tanh, np.arcsinh, np.arccosh e np.arctanh.
- Funções de valor absoluto: np.abs(), np.absolute(), np.sign().
- Funções de potência: np.power(), np.square(), np.sqrt(), np.cbrt().
- Funções de distância: np.absolute(), np.fabs(), np.hypot().
- Funções estatísticas: np.mean(), np.median(), np.std(), np.var(), np.percentile(), np.histogram().
- Funções de algebra linear: np.dot(), np.linalg.norm(), np.linalg.inv(), np.linalg.eig().

Funções de vetorização

É possível aplicar funções personalizadas aos elementos de um array usando a vetorização.
A vetorização permite que você escreva uma função em Python comum e a aplique a cada elemento de um array.
Isso é mais rápido do que o uso de laços explícitos em muitos casos.
O numpy.vectorize recebe como entrada uma função simples e retorna uma função que pode ser aplicada a arrays Numpy A função original é aplicada elemento a elemento no array de entrada, resultando em um novo array com os resultados

Funções de vetorização

Exemplo de uso

```
import numpy as np

def f(x):
    return x**2

#Vetorizando a função f
vec_f = np.vectorize(f)

#Definindo um array
arr = np.array([1, 2, 3])
#Aplicando a função vec_f ao array arr
resultado = vec_f(a)
print("Resultado: ", resultado)
#Resultado:  [1 4 9]
```

Seção 6. Criação de arrays

Função np.linspace()

Criação de um array com valores igualmente espaçados

- A função np.linspace() é utilizada para criar um array com valores igualmente espaçados dentro de um intervalo especificado.
- Sua sintaxe é a seguinte:

```
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

- start: valor de início do intervalo
- stop: valor final do intervalo
- num: número de valores igualmente espaçados a serem gerados (padrão: 50)
- endpoint: se True, inclui o valor final no array (padrão: True)
- retstep: se True, retorna o tamanho do espaçamento entre os valores gerados (padrão: False)
- dtype: tipo de dados do array (padrão: float)

Exemplo de uso da função np.linspace()

Criação de um array com valores igualmente espaçados

- Para criar um array com valores igualmente espaçados de 0 a 10 com 100 elementos, podemos fazer o seguinte:

```
import numpy as np
arr = np.linspace(0, 10, num=100)
print(arr)
```

- A saída será um array com 100 elementos igualmente espaçados de 0 a 10.

Exemplo de uso da função np.linspace()

Criação de um gráfico com base em um array criado com np.linspace()

- Podemos utilizar um array criado com np.linspace() para gerar gráficos.
- Por exemplo, podemos gerar o gráfico da função seno no intervalo de 0 a 2π da seguinte maneira:

```
import numpy as np
import matplotlib.pyplot as plt
#Criação do array de valores de x
x = np.linspace(0, 2*np.pi, num=100)
#Criação do array de valores de y
y = np.sin(x)
#Plotagem do gráfico
plt.plot(x, y)
plt.show()
```

Função np.arange()

Criação de um array com valores espaçados em um intervalo

- A função np.arange() é utilizada para criar um array com valores espaçados dentro de um intervalo especificado.
- Sua sintaxe é a seguinte:

```
np.arange(start, stop, step=1, dtype=None)
```

- start: valor de início do intervalo
- stop: valor final do intervalo (não incluso)
- step: tamanho do espaçamento entre os valores gerados (padrão: 1)
- dtype: tipo de dados do array (padrão: None)

Exemplo de uso da função np.arange()

Criação de um array com valores espaçados em um intervalo

- Para criar um array com valores espaçados de 0 a 9 com passo de 2, podemos fazer o seguinte:

```
import numpy as np
arr = np.arange(0, 10, step=2)
print(arr)
```

- A saída será um array com valores espaçados de 0 a 8 com passo de 2.

Exemplo de uso da função np.arange()

Criação de um gráfico com base em um array criado com np.arange()

- Podemos utilizar um array criado com np.arange() para gerar gráficos.
- Por exemplo, podemos gerar o gráfico da função cosseno no intervalo de 0 a 2π da seguinte maneira:

```
import numpy as np
import matplotlib.pyplot as plt
#Criação do array de valores de x
x = np.arange(0, 2*np.pi, step=0.1)
#Criação do array de valores de y
y = np.cos(x)
#Plotagem do gráfico
plt.plot(x, y)
plt.show()
```

Função np.full()

Criação de um array com um valor constante

- A função np.full() é utilizada para criar um array com um valor constante.
- Sua sintaxe é a seguinte:

```
np.full(shape, fill_value, dtype=None, order='C', *, like=None)
```

- shape: a forma (shape) do array a ser criado
- fill_value: o valor constante a ser atribuído a todos os elementos do array
- dtype: o tipo de dados do array (padrão: float64)
- order: a ordem (C ou F) em que os elementos são armazenados na memória (padrão: C)

Exemplo de uso da função np.full()

Criação de um array de zeros com forma (2, 3)

- Para criar um array de zeros com forma (2, 3), podemos fazer o seguinte:

```
import numpy as np
#Criação do array
a = np.full((2, 3), 0)
print(a)
```

- A saída será o array a, que é um array de zeros com forma (2, 3).

Exemplo de uso da função np.full()

Criação de um array de uns com forma (3, 2)

- Podemos também criar um array de uns com forma (3, 2), fazendo o seguinte:

```
import numpy as np
#Criação do array
a = np.full((3, 2), 1)
print(a)
```

- A saída será o array a, que é um array de uns com forma (3, 2).

Função np.identity()

Criação de uma matriz identidade

- A função np.identity() é utilizada para criar uma matriz identidade.
- Sua sintaxe é a seguinte:

```
np.identity(n, dtype=<class 'float'>)
```

- n: o número de linhas e colunas da matriz identidade
- dtype: o tipo de dados da matriz (padrão: float)

Exemplo de uso da função np.identity()

Criação de uma matriz identidade de ordem 3

- Para criar uma matriz identidade de ordem 3, podemos fazer o seguinte:

```
import numpy as np
#Criação da matriz identidade
A = np.identity(3)
print(A)
```

- A saída será a matriz A, que é a matriz identidade de ordem 3 com os elementos da diagonal principal iguais a 1 e os demais elementos iguais a zero.

Função np.diag()

Criação de uma matriz diagonal a partir de um array ou matriz

- A função np.diag() é utilizada para criar uma matriz diagonal a partir de um array ou matriz.
- Sua sintaxe é a seguinte:

```
np.diag(v, k=0)
```

- v: o array ou matriz a partir do qual a matriz diagonal será criada
- k: a posição da diagonal a ser extraída (padrão: 0, diagonal principal)

Exemplo de uso da função np.diag()

Criação de uma matriz diagonal a partir de um array

- Para criar uma matriz diagonal a partir de um array, podemos fazer o seguinte:

```
import numpy as np
#Criação do array
a = np.array([1, 2, 3])
#Criação da matriz diagonal
A = np.diag(a)
print(A)
```

- A saída será a matriz A, que é a matriz diagonal de ordem 3 com os elementos da diagonal principal iguais aos elementos do array a e os demais elementos iguais a zero.

Exemplo de uso da função np.diag()

Extração da diagonal de uma matriz

- Podemos também extrair a diagonal de uma matriz, fazendo o seguinte:

```
import numpy as np
#Criação da matriz
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
#Extração da diagonal
a = np.diag(A)
print(a)
```

- A saída será o array a, que contém os elementos da diagonal principal da matriz A.

Função np.tri()

Criação de uma matriz triangular inferior ou superior

- A função np.tri() é utilizada para criar uma matriz triangular inferior.
- Sua sintaxe é a seguinte:

```
np.tri(M, N=None, k=0, dtype=<class 'float'>)
```

- M: o número de linhas da matriz
- N: o número de colunas da matriz (padrão: None)
- k: o índice da diagonal a ser preenchida (padrão: 0, diagonal principal)
- dtype: o tipo de dados da matriz (padrão: float)

Exemplo de uso da função np.tri()

Criação de uma matriz triangular inferior

```
import numpy as np
A = np.tri(4) #Matriz triangular inferior 4x4
B = np.tri(4, k=1) #Matriz triangular inferior 4x4, com a diagonal principal
#deslocada uma diagonal para cima.
C = np.tri(4, k=-1) #Matriz triangular inferior 4x4, com a diagonal principal
#deslocada uma diagonal para baixo.
C = np.tri(4, 6, k=-1) #Matriz triangular inferior 4x6, com a diagonal principa
#deslocada uma diagonal para baixo.
```

Exemplo de uso da função np.tri()

Criação de uma matriz triangular superior

- Podemos também criar uma matriz triangular superior, fazendo a transposta da triangular inferior.

```
import numpy as np
A = np.tri(3, k=1) #Criação da matriz triangular inferior
B = A.T #A.T é a transposta da matriz A.
#Logo B é uma matriz triangula superior.
print(A)
```

Função np.tile()

Criação de uma cópia de um array em uma ou mais dimensões

- A função np.tile() é utilizada para criar uma ou mais cópias de um array em uma ou mais dimensões.
- Sua sintaxe é a seguinte:

```
np.tile(A, reps)
```

- A: o array a ser copiado
- reps: um inteiro ou uma tupla de inteiros que especifica o número de cópias em cada dimensão

Exemplo de uso da função np.tile()

Criação de uma matriz com cópias de um array

- Para criar uma matriz com 3 cópias de um array de 2x2, podemos fazer o seguinte:

```
import numpy as np
#Criação do array de origem
A = np.array([[1, 2], [3, 4]])
Criação da matriz com cópias do array
B = np.tile(A, (3, 1))
print(B)
```

- A saída será a matriz B, que contém 3 cópias do array A empilhadas verticalmente.

Exemplo de uso da função np.tile()

Criação de uma matriz com cópias de um array

- Podemos também criar uma matriz com 2 cópias do array de origem empilhadas horizontalmente:

```
import numpy as np
#Criação do array de origem
A = np.array([[1, 2], [3, 4]])
Criação da matriz com cópias do array
B = np.tile(A, (1, 2))
print(B)
```

- A saída será a matriz B, que contém 2 cópias do array A empilhadas horizontalmente.

Função np.concatenate()

Concatenação de dois ou mais arrays ao longo de uma ou mais dimensões

- A função np.concatenate() é utilizada para concatenar dois ou mais arrays ao longo de uma ou mais dimensões.
- Sua sintaxe é a seguinte:

```
np.concatenate((a1, a2, ...), axis=0)
```

- a1, a2, ...: os arrays a serem concatenados
- axis: a dimensão ao longo da qual os arrays serão concatenados (padrão: 0)

Exemplo de uso da função np.concatenate()

Concatenação de dois arrays

- Para concatenar dois arrays ao longo da primeira dimensão, podemos fazer o seguinte:

```
import numpy as np
#Criação dos arrays
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
#Concatenação dos arrays
a3 = np.concatenate((a1, a2))
print(a3)
```

- A saída será o array [1, 2, 3, 4, 5, 6], que é a concatenação dos arrays a1 e a2 ao longo da primeira dimensão.

Exemplo de uso da função np.concatenate()

Concatenação de dois arrays em uma nova dimensão

- Podemos também concatenar os dois arrays em uma nova dimensão, fazendo o seguinte:

```
import numpy as np
#Criação dos arrays
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
#Concatenação dos arrays em uma nova dimensão
a3 = np.concatenate((a1[:, np.newaxis], a2[:, np.newaxis]), axis=1)
print(a3)
```

- A saída será o array [[1, 4], [2, 5], [3, 6]], que é a concatenação dos arrays a1 e a2 em uma nova dimensão.

Função np.stack()

Empilhamento de arrays ao longo de uma nova dimensão

- A função np.stack() é utilizada para empilhar dois ou mais arrays ao longo de uma nova dimensão.
- Sua sintaxe é a seguinte:

```
np.stack(arrays, axis=0)
```

- arrays: os arrays a serem empilhados (pode ser uma tupla ou uma lista de arrays)
- axis: a dimensão ao longo da qual os arrays serão empilhados (padrão: 0)

Exemplo de uso da função np.stack()

Empilhamento de dois arrays em uma nova dimensão

- Para empilhar dois arrays na direção horizontal, fazemos o seguinte:

```
import numpy as np
#Criação dos arrays
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
#Empilhamento dos arrays em uma nova dimensão
a3 = np.stack((a1, a2))
print(a3)
# Saída:
#[[1 2 3]
# [4 5 6]]
```

Exemplo de uso da função np.stack()

Empilhamento de dois arrays em uma nova dimensão

- Ou então,

```
import numpy as np
#Criação dos arrays
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
#Empilhamento dos arrays em uma nova dimensão
a3 = np.stack((a1, a2), axis=0)
print(a3)
# Saída:
#[[1 2 3]
# [4 5 6]]
```

Exemplo de uso da função np.stack()

Empilhamento de dois arrays em uma nova dimensão

- Para empilhar dois arrays na direção vertical, podemos fazer o seguinte:

```
import numpy as np
#Criação dos arrays
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
#Empilhamento dos arrays em uma nova dimensão
a3 = np.stack((a1, a2), axis=1)
print(a3)
```

- A saída será o array [[1, 4], [2, 5], [3, 6]], que é o empilhamento dos arrays a1 e a2 em uma nova dimensão.

Exemplo de uso da função np.stack()

Empilhamento de três arrays em uma nova dimensão

- Podemos também empilhar três arrays. Por exemplo:

```
import numpy as np
#Criação dos arrays
a1 = np.array([1, 2, 3])
a2 = np.array([4, 5, 6])
a3 = np.array([7, 8, 9])
#Empilhamento dos arrays em uma nova dimensão
a4 = np.stack((a1, a2, a3), axis=1)
print(a4)
```

- A saída será o array [[1, 4, 7], [2, 5, 8], [3, 6, 9]], que é o empilhamento dos arrays a1, a2 e a3 em uma nova dimensão.

Seção 7. Cuidados ao lidar com arrays

Cuidados ao lidar com arrays

Evitar o uso de variáveis globais

- O uso de variáveis globais pode levar a resultados inesperados em operações com arrays.
- É recomendado o uso de variáveis locais em funções e a passagem de arrays como argumentos.

```
#Exemplo de uso de variável global
import numpy as np
a = np.array([1, 2, 3])
def func():
    global a
    a = a + 1
func()
print(a)
#Saída: array([2, 3, 4])
```

Cuidados ao lidar com arrays

Cópia versus visualização de arrays

- A operação de atribuição de um array a uma nova variável cria uma visualização (view) do array original, não uma cópia.
- Alterações na visualização afetam o array original, e vice-versa.
- É recomendado o uso da função np.copy() para criar cópias de arrays.

```
#Exemplo de criação de uma visualização
import numpy as np
a = np.array([1, 2, 3])
b = a
b[0] = 0
print(a)
#Saída: array([0, 2, 3])
```

Cuidados ao lidar com arrays

Operações com arrays de formas diferentes

- Operações com arrays de formas diferentes podem levar a resultados inesperados.
- É recomendado o uso de funções de redimensionamento (reshape, resize) ou de transposição (transpose, T) para igualar as formas dos arrays antes da operação.

```
#Exemplo de operação com arrays de formas diferentes
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([[1, 2], [3, 4]])
c = a + b
print(c)
#Saída: ValueError: operands could not be broadcast together with shapes (4,) (
```

Verificar a precisão numérica dos elementos do array

Funções para verificação de precisão

- As seguintes funções do NumPy podem ser usadas para verificar a precisão dos elementos de um array:
  - np.isfinite() - verifica se os valores são finitos (não são NaN ou infinitos).
  - np.isinf() - verifica se os valores são infinitos.
  - np.isnan() - verifica se os valores são NaN (Not a Number).
  - np.issubdtype() - verifica se os valores têm um tipo de dados específico (por exemplo, se são inteiros ou floats).

```
#Exemplo de uso das funções de verificação de precisão
import numpy as np
a = np.array([1.0, 2.0, np.inf, np.nan])
print(np.isfinite(a)) # Saída: [ True True False False]
print(np.isinf(a)) # Saída: [False False True False]
print(np.isnan(a)) # Saída: [False False False True]
print(np.issubdtype(a.dtype, np.float)) # Saída: True
```

Cuidados ao lidar com elementos nulos (NaN) ou infinitos (Inf) no array

Verificar a presença de elementos nulos (NaN)

- A presença de elementos nulos (NaN) em um array pode afetar o resultado de operações matemáticas e estatísticas.
- É recomendado verificar a presença de elementos nulos com a função np.isnan() e substituí-los por um valor adequado.

```
#Exemplo de verificação de elementos nulos
import numpy as np
a = np.array([1, 2, np.nan, 4, 5])
#Verificação de elementos nulos
mask = np.isnan(a)
a[mask] = 0
print(a)
#Saída: array([1., 2., 0., 4., 5.])
```

Cuidados ao lidar com elementos nulos (NaN) ou infinitos (Inf) no array

Verificar a presença de elementos infinitos (Inf)

- A presença de elementos infinitos (Inf) em um array pode afetar o resultado de operações matemáticas e estatísticas.
- É recomendado verificar a presença de elementos infinitos com a função np.isinf() e substituí-los por um valor adequado.

```
#Exemplo de verificação de elementos infinitos
import numpy as np
a = np.array([1, 2, np.inf, 4, 5])
#Verificação de elementos infinitos
mask = np.isinf(a)
a[mask] = 0
print(a)
#Saída: array([1., 2., 0., 4., 5.])
```

Cuidados ao lidar com elementos nulos (NaN) ou infinitos (Inf) no array

Remoção de elementos nulos ou infinitos

- A presença de elementos nulos (NaN) ou infinitos (Inf) em um array pode ser um problema em algumas operações.
- É possível remover os elementos nulos ou infinitos com as funções np.isnan() e np.isinf(), respectivamente.

```
#Exemplo de remoção de elementos nulos e infinitos
import numpy as np
a = np.array([1, 2, np.nan, 4, np.inf, 5])
#Remoção de elementos nulos e infinitos
mask = np.logical_or(np.isnan(a), np.isinf(a))
a = a[~mask]
print(a)
#Saída: array([1., 2., 4., 5.])
```

Seção 8. Módulo random do numpy

Módulo random do numpy

Funções mais utilizadas

O módulo random do numpy é usado para gerar números aleatórios de diferentes distribuições e formas. Algumas das funções mais importantes do módulo são:

- rand(): gera números aleatórios de uma distribuição uniforme entre 0 e 1.
- random(): gera números aleatórios de uma distribuição uniforme entre 0 e 1 (similar a rand()).
- randn(): gera números aleatórios de uma distribuição normal padrão (média 0 e desvio padrão 1).
- uniform(): gera números aleatórios de uma distribuição uniforme contínua entre um limite inferior e superior dados.
- randint(): gera números inteiros aleatórios entre um limite inferior e superior dados.
- choice(a, size): escolhe elementos aleatórios de um array ou intervalo dado.
- shuffle(): embaralha os elementos de um array em ordem aleatória.
- permutation(): retorna uma permutação aleatória de um array ou intervalo dado.
- seed(): define a semente do gerador de números aleatórios para reproduzir os mesmos resultados.

Função numpy.random.rand

A função rand() do numpy retorna um array de amostras aleatórias de uma distribuição uniforme entre 0 e 1. Ela permite especificar as dimensões do array como argumentos e retorna um array da forma desejada. Se não fornecermos nenhum argumento, ela retornará um valor float.
Por exemplo:

- numpy.random.rand() retorna um valor float entre 0 e 1.
- numpy.random.rand(3) retorna um array de uma dimensão com 3 elementos entre 0 e 1.
- numpy.random.rand(2,4) retorna um array de duas dimensões com 2 linhas e 4 colunas entre 0 e 1.

Função numpy.random.random

A função numpy.random.random do numpy é uma função que retorna um array de números aleatórios de uma distribuição uniforme entre 0 e 1. Ela aceita um argumento size que especifica a forma do array retornado. Se nenhum argumento for dado, ela retornará um float. Por exemplo:

- numpy.random.random() retorna um valor float entre 0 e 1.
- numpy.random.random(3) retorna um array de uma dimensão com 3 elementos entre 0 e 1.
- numpy.random.random((2,4)) retorna um array de duas dimensões com 2 linhas e 4 colunas entre 0 e 1.

Função numpy.random.random

Diferença entre numpy.random.rand e numpy.random.random

A principal diferença entre np.random.rand e np.random.random é a forma como eles aceitam os argumentos para especificar o tamanho da matriz de saída. O np.random.rand aceita argumentos separados para cada dimensão, enquanto o np.random.random aceita um único argumento de tupla. Por exemplo, para criar uma matriz de amostras com forma (3, 5), você pode escrever:

```
import numpy as np
arr1 = np.random.rand(3, 5)
arr2 = np.random.random((3, 5))
```

Ambas as funções geram amostras aleatórias de uma distribuição uniforme no intervalo [0, 1).

Função numpy.random.randn

A função numpy.random.randn do numpy cria um array de forma especificada e o preenche com valores aleatórios de acordo com a distribuição normal padrão. Você pode fornecer argumentos positivos para definir a forma do array (d0, d1, ..., dn).
Por exemplo, numpy.random.randn(2, 3) gera um array de 2 linhas e 3 colunas com números aleatórios da distribuição normal padrão.

Função numpy.random.uniform

A função numpy.random.uniform do numpy é usada para gerar amostras de uma distribuição uniforme. Isso significa que qualquer valor dentro do intervalo dado tem a mesma probabilidade de ser sorteado. Você pode especificar os limites inferior e superior do intervalo (low e high) e o tamanho da amostra (size) como parâmetros da função. Por exemplo:

```
import numpy as np
# Gerar 10 amostras entre 0 e 5
x = np.random.uniform(low=0, high=5, size=10)
print(x)
```

Função numpy.random.randint

A função numpy.random.randint é usada para gerar números inteiros aleatórios de uma distribuição uniforme discreta no intervalo [a, b). Se b for None (o padrão), os resultados serão de [0, a). Você pode especificar o tamanho e o tipo de dados dos números gerados como parâmetros opcionais.

Função numpy.random.randint

Função numpy.random.randint

```
import numpy as np
# Gerar um número inteiro entre 0 e 9
arr1 = np.random.randint(10)
# Gerar um array 2D de 3x4 com números inteiros entre 1 e 100
arr2 = np.random.randint(1, 101, size=(3,4))
# Gerar um array 1D de 5 com números inteiros de 8 bits entre -128 e 127
arr3 = np.random.randint(-128, 128, size=5, dtype=np.int8)
# Gerar um array booleano de tamanho 5
arr4 = np.random.randint(2, size=5, dtype=bool)
```

Função numpy.random.choice

A função numpy.random.choice gera uma amostra aleatória a partir de um array 1-D ou de uma lista ou de uma tupla. Você pode especificar o tamanho da amostra, se a amostragem é com ou sem reposição, as probabilidades associadas a cada elemento e o eixo ao longo do qual a amostra é gerada.

Função numpy.random.choice

Exemplos

Para gerar uma amostra de 5 elementos do array [1, 2, 3, 4] com reposição e probabilidades uniformes, você pode escrever:

```
arr = np.random.choice([1, 2, 3, 4], size=5)
```

Função numpy.random.choice

Exemplos

Para gerar uma amostra de 10 elementos do intervalo [0, 20) sem reposição e com probabilidades proporcionais aos números pares, você pode escrever:

```
pesos = np.array([0.5 if i % 2 == 0 else 0 for i in range(20)])
amostra = np.random.choice(20, size=10, replace=False, p=pesos/np.sum(pesos))
```

Função numpy.random.choice

Exemplos

Para gerar uma amostra de 3 linhas do array bidimensional [[1, 2], [3, 4], [5, 6], [7, 8]] com reposição e probabilidades uniformes, você pode escrever:

```
array = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
amostra = np.random.choice(array.shape[0], size=3)
amostra = array[amostra]
```

Função numpy.random.choice

Exemplos

Para gerar uma amostra de caracteres da string "abcdefg"com reposição e probabilidades personalizadas para cada caractere (por exemplo: a tem probabilidade de 0.1 e g tem probabilidade de 0.4), você pode escrever:

```
string = "abcdefg"
probabilidades = [0.1, 0.05, 0.15, 0.1 ,0.2 ,0.05 ,0.4]
amostra = np.random.choice(list(string), size=5,p=probabilidades)
amostra = "".join(amostra)
```

Função numpy.random.shuffle

Introdução

A função `numpy.random.shuffle` é uma função da biblioteca numpy que embaralha os elementos de uma lista ou matriz de forma aleatória.

Função numpy.random.shuffle

Exemplo

Considere a lista a abaixo:

```
import numpy as np

a = [1, 2, 3, 4, 5]
np.random.shuffle(a)
print(a)
```

A saída desse código será uma permutação aleatória dos elementos da lista a.

Função numpy.random.shuffle

Parâmetros

Função numpy.random.shuffle

Observações

- A função `numpy.random.shuffle` não possui parâmetros adicionais.
- A função `numpy.random.shuffle` altera a lista original. Se for necessário preservar a lista original, é preciso fazer uma cópia da lista antes de embaralhá-la.

Função numpy.random.shuffle

Exemplo

Considere a lista a abaixo:

```
import numpy as np

a = [1, 2, 3, 4, 5]
b = a.copy()
np.random.shuffle(b)
print(a)
print(b)
```

A saída desse código será a lista a na ordem original e a lista b embaralhada aleatoriamente.

Função numpy.random.permutation

Introdução

A função `numpy.random.permutation` retorna uma permutação aleatória de um array.

Função numpy.random.permutation

Exemplo 1

```
import numpy as np
x = np.array([1, 2, 3, 4, 5])
print(np.random.permutation(x))
```

A saída será um array com os elementos permutados de forma aleatória, por exemplo: `[2, 5, 1, 4, 3]`.

Função numpy.random.permutation

Exemplo 2

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.random.permutation(x))
```

A saída será um array com as linhas permutadas de forma aleatória, por exemplo: `[[3, 4], [1, 2]]`.

Função numpy.random.seed

Definição e Uso

A função `numpy.random.seed` é usada para definir a semente (seed) para o gerador de números aleatórios do NumPy. A semente é um número inteiro que é usado como ponto de partida para o algoritmo de geração de números aleatórios.
Ao definir a semente, podemos garantir que as sequências de números aleatórios geradas pelo numpy sejam reproduzíveis. Ou seja, se usarmos a mesma semente em diferentes execuções do código, obteremos a mesma sequência de números aleatórios.
Exemplo de uso:

```
import numpy as np
np.random.seed(42)
print(np.random.rand(3)) # [0.37454012 0.95071431 0.73199394]
np.random.seed(12)
print(np.random.rand(3)) # [0.15416284 0.7400497  0.26331502]
np.random.seed(42)
print(np.random.rand(3)) # [0.37454012 0.95071431 0.73199394]
```

Função numpy.random.seed

Observações

Algumas observações sobre a função `numpy.random.seed`:

- A semente pode ser qualquer número inteiro.
- Se não definirmos a semente explicitamente, o NumPy usará uma semente padrão que varia de acordo com o sistema operacional e a versão do NumPy.
- Uma vez que a semente é definida, todas as funções de geração de números aleatórios do NumPy (por exemplo, `numpy.random.rand`, `numpy.random.randn`, `numpy.random.randint`) usarão essa mesma semente.
- Se mudarmos a semente durante a execução do código, as sequências de números aleatórios geradas a partir desse ponto serão diferentes das sequências geradas anteriormente.

Referências

Referências I

- NUMPY quickstart. Accessado em 28 de fevereiro de 2023. Disponível em: <https://numpy.org/doc/stable/user/quickstart.html>.
- NUMPY User Guide. Accessado em 28 de fevereiro de 2023. Disponível em: <https://numpy.org/doc/1.23/numpy-user.pdf>.