*Programação com arrays: introdução ao NumPy*

Estruturas de dados. Arrays

Estruturas de Dados.

Normalmente, quando trabalhamos no Python com dados orientados por Data Science, vemos que estes vêm em diferentes formatos ou tipos. Estes conjuntos de dados são organizados de uma certa forma e, graças a isto, podemos acessar estes dados, operar com eles, mostrá-los e, de modo geral, manipulá-los. Como vimos na última aula, vamos falar aqui sobre **estruturas de dados**. Não vamos pensar apenas em dados que poderíamos ver, por exemplo, em uma planilha. Esta análise pode ser estendida a dados de outros tipos, tais como arquivos de áudio ou imagens. Estes diversos tipos de dados têm que ser transformados em números para poder ser interpretados pelos algoritmos.

Por outro lado, para trabalhar com estruturas de dados é necessário realizar operações que são habituais para este tipo de trabalho. O Python já dispõe de uma série de bibliotecas especializadas para esses trabalhos. Nesta e na próxima aula veremos os dois pacotes principais: NumPy (Numerical Python) e Pandas. Estes pacotes oferecem funções avançadas para trabalhar com conjuntos de dados de tamanhos variados. De uma certa forma, é semelhante a trabalhar com listas, como vimos na aula anterior, mas nestes casos o armazenamento é mais eficiente, e funciona muito bem com grandes conjuntos de dados.

Arrays

Vamos relembrar da Aula 02 que no Python existem vários tipos de estruturas de dados, tais como list, tuple, dict e set. Cada uma dessas estruturas apresenta algumas características particulares. A realidade é que geralmente estas características não são suficientes para atender às necessidades do data scientist.

A seguir, estenderemos a aplicação desses tipos de estrutura de dados, adicionando o tipo de dados array.

Comparado com o tipo list, ambos os tipos de dados servem para armazenar conjuntos de dados ordenados na memória. No entanto, eles têm algumas diferenças. Enquanto o tipo list pode armazenar dados de diferentes tipos, o array armazena dados de um único tipo. Isto lhe permite ser mais eficiente, especialmente ao trabalhar com grandes conjuntos de dados. Além disso, para definir um dado do tipo array, devemos previamente importar o módulo array. Vejamos sua aplicação em código:

| L = list(range(10)) L # Mostra uma lista de 10 elementos import array as arr #Precisamos importar o módulo array para criar arrays A = arr.array("i",range(10)) A # Mostra um array de 10 elementos |
| --- |

NumPy e ndarrays. Criação de ndarrays

Como conseguir uma melhoria no uso dessas estruturas? O pacote NumPy nos dá a resposta. O NumPy define um novo tipo de dado denominado ndarray, que funciona como um array com vitaminas. Vamos começar criando uma estrutura deste tipo:

| import numpy as np # importamos a biblioteca NumPy Npa = np.array(range(10)) # criamos um ndarray de 10 elementos Npa |
| --- |

Também temos funções predeterminadas para criar ndarrays já preenchidos com dados. Mostramos alguns deles a seguir:

| Np\_zero = np.zeros(10) # Ndarray de zeros Np\_zero\_int = np.zeros(10, dtype=int) # Ndarray de zeros, mas especificando que são inteiros Np\_um = np.ones(10) # Ndarray de uns print(Np\_zero, Np\_zero\_int, Np\_um) Np\_completo = np.full(10,256) # Preencher um array de 10 elementos com o número 256 Np\_completo\_dimensoes = np.full((2,3),256) # Elaborar uma matriz de 2 linhas por 3 colunas e preenchê-la com o número 256 print(Np\_completo) print(Np\_completo\_dimensoes) Np\_arranjo = np.arange(10) # Elaborar de forma compacta um array com números ordenados Np\_random\_dimensoes = np.random.randint(10, size=(3, 4)) # Array random de 3 linhas por 4 colunas print(Np\_random\_dimensoes) |
| --- |

Desta forma podemos criar arrays no NumPy.

Tipos de dados e atributos de arrays

Tipos de dados

Os objetos do tipo ndarray podem ser de apenas um tipo. Como vimos anteriormente, podemos deixar o tipo vazio, com o qual o Python decide o tipo baseando-se nos elementos inseridos; ou também podemos especificar o tipo, por exemplo, quando sabemos a priori que tipo de dados o ndarray conterá, de forma a tornar mais eficiente o uso da memória.

Para especificar o tipo de dado na criação do ndarray, podemos consultar os tipos na lista oficial publicada neste link: <https://numpy.org/devdocs/user/basics.types.html>

Atributos

Sendo objetos do Python, os ndarrays têm atributos. Estes atributos cuidam de manter o registro do tamanho, forma, consumo de memória e tipo de dados do ndarray.

| print(Np\_zero.ndim) # Vetor unidimensional print(Np\_completo\_dimensoes.ndim) # Matriz de 2 dimensões (linhas e colunas) print(Np\_completo\_dimensoes.shape) # 2 linhas por 3 colunas print(Np\_completo\_dimensoes.size) # 6 elementos no total print(Np\_zero.dtype) # Ponto flutuante print(Np\_zero\_int.dtype) # Inteiros print(Np\_zero.itemsize) # Tamanho de cada elemento print(Np\_zero.nbytes) # Tamanho total do array print(Np\_zero\_int.nbytes) # Os inteiros ocupam menos espaço do que os elementos de ponto flutuante |
| --- |

Indexação e acesso

Acesso aos elementos

Anteriormente dissemos que os arrays e ndarrays, como as estruturas do tipo list, contêm dados ordenados. Isto significa que a ordem de cada elemento é sempre a mesma, portanto, podemos nos referir a um elemento (ou *referenciá-lo*) a partir de seu índice ou número de posição. Como uma particularidade, no Python, os índices começam a partir do número 0. Com o qual um conjunto de 10 elementos terá índices de 0 a 9. Vejamos um exemplo:

| arranjo = range(1,11) # A função range inclui o primeiro número e exclui o último  Np\_dez\_numeros = np.array(arranjo) # Criar um ndarray que contenha a faixa gerada print(Np\_dez\_numeros) print(Np\_dez\_numeros[0]) # Primeiro elemento print(Np\_dez\_numeros[4]) # Quinto elemento print(Np\_dez\_numeros[-1]) # Contagem em reverso, começando com o -1 (último elemento) print(Np\_dez\_numeros[-2]) # Penúltimo elemento print(Np\_random\_dimensoes) print(Np\_random\_dimensoes[2,1]) #Terceira linha, segunda coluna |
| --- |

Acesso aos subarrays

No Python, podemos "recortar" porções em conjuntos de dados usando a operação de *slice*. Para isso, usamos a seguinte notação: objeto[a partir de:até:tamanho\_da\_array]. Por padrão, os valores são percorridos de 0 até o tamanho do objeto, de um passo (um elemento de cada vez). Vejamos alguns exemplos a seguir:

| print(Np\_dez\_numeros[:4]) # Primeiros quatro elementos print(Np\_dez\_numeros[3:]) # Desde o quarto elemento até o final print(Np\_dez\_numeros[4:7]) # Cuidado! Desde o quinto elemento até o sétimo  print(Np\_dez\_numeros[::2]) # Desde o primeiro elemento, saltando aos pares print(Np\_dez\_numeros[1::2]) # Desde o segundo elemento, saltando aos pares print(Np\_dez\_numeros[::-2]) # Se o step for negativo, caminho de volta |
| --- |

No caso de haver mais de uma dimensão, acessamos de maneira ordenada cada dimensão da forma objeto[dimensão\_1,dimensão\_2,...]. Em cada dimensão, podemos fazer um slice da mesma forma que fazíamos anteriormente.

| print(Np\_random\_dimensoes[2,]) # Terceira linha, todas as colunas print(Np\_random\_dimensoes[:2,:2]) # Subconjunto das primeiras duas linhas e primeiras duas colunas print(Np\_random\_dimensoes[2,3]) # Terceira linha, quarta coluna |
| --- |

Operações básicas: reshape, concatenação, splitting

Existem muitas operações para poder explorar ao máximo o potencial que oferece o NumPy. Podemos organizar as principais operações em três grandes grupos: operações básicas, agregações e operações vetorizadas. Existem vários outros tipos de operações, mas aqui veremos as básicas que nos permitem entrar no mundo do Data Science.

Reshape

Até agora, não vimos uma maneira rápida de organizar os elementos em um array multidimensional. Suponhamos, por exemplo, que queremos preencher um tabuleiro de xadrez (8 x 8 casas) com números, e temos à mão a função range() que nos permite gerar os números sem necessidade de escrevê-los à mão.

Se tentarmos fazer um tabuleiro de xadrez cheio de zeros, ou qualquer número fixo, com o que sabemos até agora, é possível fazê-lo facilmente.

| Xadrez\_zeros = np.zeros((8,8)) print(Xadrez\_zeros) Xadrez\_num = np.full((8,8),35) print(Xadrez\_num) |
| --- |

Por outro lado, se quisermos preenchê-lo com números consecutivos de 1 a 64, nos dará um erro.

| arranjo\_xadrez = range(1,65) Xadrez = np.array((8,8),arranjo\_xadrez) # Isto não funciona |
| --- |

Para solucionar esta e outras situações similares, o NumPy define uma operação denominada *reshaping*. Basicamente, tomamos um ndarray como entrada e o "reformulamos", em termos de dimensões. Vejamos alguns exemplos:

| Xadrez\_64 = np.arange(1,65) # Um array de números de 1 a 64  print(Xadrez\_64) Xadrez\_64 = np.arange(1,65).reshape(8,8) # Organizá-lo em 8 x 8 print(Xadrez\_64) Xadrez\_64 = Xadrez\_64.reshape(8,8) # Equivalente ao comando anterior print(Xadrez\_64) |
| --- |

Concatenação

Para "vincular" ou concatenar dois ou mais arrays, utilizamos as funções concatenate e vstack / hstack. Vejamos alguns exemplos:

| Array\_1 = np.random.randint(10,size=5) Array\_2 = np.random.randint(10,size=5) print(Array\_1, Array\_2) # Dois arrays de 5 elementos, com números aleatórios do 0 ao 9 Arrays\_concatenados = np.concatenate([Array\_1, Array\_2]) # Um array que contenha os 10 números concatenados print(Arrays\_concatenados) |
| --- |

Se quisermos combinar explicitamente os arrays na forma horizontal ou verticalmente, usamos as funções vstack y hstack

| Array\_1 = np.random.randint(10,size=5) Array\_2 = np.random.randint(10,size=5) print(Array\_1, Array\_2) # Dois arrays de 5 elementos, com números aleatórios do 0 ao 9 Arrays\_concatenados = np.concatenate([Array\_1, Array\_2]) # Um array que contenha os 10 números concatenados print(Arrays\_concatenados) Array\_extra = np.array([[10],[20]]) # Um array de duas dimensões (vertical) com os números 10 e 20 print(Array\_extra) Array\_apilados\_v\_h = np.hstack([Array\_extra,Array\_apilados\_v]) # Concatenar o array vertical com o array empilhado print(Array\_apilados\_v\_h) print(np.shape(Array\_apilados\_v\_h)) # Quantas linhas e colunas ele tem? |
| --- |

Desta forma, podemos combinar arrays.

Splitting

Por outro lado, podemos "dividir" os arrays utilizando as funções split, hsplit y vsplit. Estas funções recebem como argumento o array a "ser dividido" e uma lista de pontos de corte. Podemos pensar nos pontos de corte como as "lacunas" entre os elementos do array. Vejamos os exemplos a seguir:

| Array\_partido = np.split(Arrays\_concatenados,[2]) # Divide o array entre o segundo e o terceiro elemento print(Array\_partido) Array\_partido\_2 = np.split(Arrays\_concatenados,[2,8]) # Divide o array entre o segundo e o terceiro, e entre o oitavo e o nono elemento print(Array\_partido\_2) Parte\_1, Parte\_2, Parte\_3 = Array\_partido\_2 # Armazena cada elemento em uma variável. Muito útil print(Parte\_1, Parte\_2, Parte\_3) Ajedrez\_partido\_1 = np.hsplit(Ajedrez\_64,[4]) # Divide o tabuleiro de xadrez ao meio na vertical print(Ajedrez\_partido\_1) Ajedrez\_partido\_2 = np.vsplit(Ajedrez\_64,[4]) # Divide o tabuleiro de xadrez ao meio na horizontal print(Ajedrez\_partido\_2) |
| --- |

Desta forma, podemos trabalhar com operações básicas sobre arrays.

Agregações

Como faríamos para calcular uma média de dez números dentro de um array? Simples, somamos os números e dividimos o resultado por 10. Vamos fazer isso no Python:

| Array\_aleatorio = np.random.randint(10,size=10) print(Array\_aleatorio) suma = 0 for i in Array\_aleatorio:  suma += i promedio = suma / np.size(Array\_aleatorio) print(promedio) |
| --- |

Esta resolução é bonita e elegante. No entanto, é possível que precisemos calcular frequentemente este tipo de medidas resumidas, tais como a média, a variância, o desvio, a mediana, e similares. Para estes casos frequentes, o NumPy possui atalhos que são muito úteis na hora de realizar os cálculos.

| print(Array\_aleatorio.sum()) # Soma dos elementos print(Array\_aleatorio.mean()) # Média dos elementos print(Array\_aleatorio.max()) # Elemento maior. print(np.median(Array\_aleatorio)) # Média print(np.std(Array\_aleatorio)) # Desvio padrão print(np.var(Array\_aleatorio)) # Variância |
| --- |

Todas as funções podem ser escritas como np.nombre\_de\_função(objeto). As funções mais comuns (sum, mean, max), podem ser escritas de forma equivalente como métodos aplicados sobre o objeto (por exemplo, objeto.mean().

Desta forma, o NumPy organiza as funções de agregação, ou resumo, de forma compacta e em uma única linha de código.

É necessário esclarecer que o Python oferece estas funções por padrão, mas no NumPy elas são otimizadas para trabalhar com grandes volumes de dados.

Operações vetorizadas

Finalmente, o NumPy traz um conjunto de funções para operações otimizadas para arrays, de forma a trabalhar essas estruturas da maneira de vetores ou matrizes matemáticas. Estas funções são chamadas operações vetorizadas, ou funções universais (ufuncs). Muitas vezes é necessário aplicar operações sobre cada elemento, como por exemplo, somar uma constante ou adicionar dois vetores elemento por elemento, ou realizar um produto matricial. Para isso, podemos operar utilizando estas funções com os ndarrays do NumPy. Vejamos alguns exemplos:

| print(Array\_1, Array\_2)  print(Array\_1 + 5) # Soma 5 a cada elemento do Array\_1 print(Array\_1 + Array\_2) # Soma o Array\_1 y Array\_2, elemento por elemento print(np.add(Array\_1,Array\_2)) # Igual ao anterior print(np.matmul(Array\_1, Array\_2)) # Produto vetorial: São multiplicados os elementos na mesma posição do Array\_1 e Array\_2 ( multiplicam-se os dois primeiros, depois os dois segundos, etc.), e então todos os resultados são somados. |
| --- |

O guia completo de referência das ufuncs está disponível em https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

Sempre que se trabalha com os ndarrays de NumPy é recomendável utilizar estas funções, pois além de serem mais compactas do que realizar a programação manualmente, elas são construídas de tal forma que funcionam mais rapidamente. Isto é especialmente importante quando utilizamos grandes conjuntos de dados.

Exercícios

1. Gerar um array aleatório de 100 elementos. Calcular a mediana correspondente.
2. Relembrar os exercícios com funções para cálculo de fatorial e soma de séries. Repetir os dois exercícios (pode ser sobre o mesmo código), mas agora utilizar as novas operações aprendidas com os ndarrays.