**02 - Arrays NumPy**

Nesta última seção do curso conversaremos sobre o pacote **Numpy**, uma abreviação de *Numerical Python*. Este é um dos pacotes mais importantes para processamento numérico no Python, além de ser a base para a maioria dos pacotes científicos que utilizam dados numéricos. No notebook deixamos inclusive um texto explicativo que introduz a esse pacote.

Alguns pontos importantes são a disponibilidade de um poderoso objeto array multidimensional, as funções matemáticas sofisticadas para operações com arrays sem a necessidade de utilização de laços for, e os recursos de algebra linear e geração de números aleatórios. O Numpy também serve como um contêiner eficiente para transporte de dados multidimensionais entre algoritmos e pacotes com os quais trabalhamos, além de servir de base para construção de outros pacotes.

Existem diversos pacotes do Python que não são distribuídos na versão padrão (*default*) do seu interpretador, como o próprio Numpy, o Pandas (assunto do próximo curso), o Scikit-learn, o Matplotlib e o Seaborn. O Google Colab disponibiliza esses pacotes previamente, eliminando a necessidade de sua instalação. Caso você esteja usando outra distribuição, a instalação pode ser necessária.

Para utilizarmos os métodos e funções de interesse em cada pacote, é necessário importá-los. No caso do Numpy, basta usarmos a palavra-chave import (reservada do Python) seguida do nome dessa biblioteca.

import numpyCOPIAR CÓDIGO

Na [documentação do Numpy](https://numpy.org/doc/1.16/reference/generated/numpy.arange.html), encontramos algumas explicações sobre o arange(), que basicamente funciona como a função range() - ou seja, ela cria uma lista (nesse caso um array numpy) que se inicia no 0 e tem o tamanho indicado por parâmetro.

Para chamarmos essa função, usaremos numpy.arange(), passando como valor o número 10. Como retorno, teremos um array de tamanho 10:

numpy.arange(10)COPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Existe também outra forma de fazer essa importação, que é atribuindo um apelido à biblioteca, algo que fizemos no início desse curso.

import numpy as npCOPIAR CÓDIGO

Esse tipo de importação é muito comum em livros e textos sobre data science, e se estende também a outras bibliotecas. A Pandas, por exemplo, costuma ser apelidada de pd. Com a atribuição do apelido, poderemos chamar a função arange() da seguinte forma:

np.arange(10)COPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Se conhecemos bem a biblioteca e não queremos encher a memória do computador com informações que não serão utilizadas, podemos importar somente as funções que iremos utilizar. Para isso, basta usarmos a instrução from seguida do nome da biblioteca, a palavra-chave import e a função desejada - nesse caso, arange().

from numpy import arangeCOPIAR CÓDIGO

Com isso teremos acesso à função arange\*() diretamente.

arange(10)COPIAR CÓDIGO

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Agora começaremos a criar propriamente os famosos **arrays Numpy**. Uma das maneiras de fazermos isso é a partir de uma lista. Para isso, criaremos uma variável km que receberá a chamada de np.array(). Passaremos como parâmetro dessa função uma lista contendo números aleatórios.

km = np.array([1000, 2300, 4987, 1500])

kmCOPIAR CÓDIGO

array([1000, 2300, 4987, 1500])

Também é possível criar arrays numpy a partir de dados externos, algo que já fizemos no início desse curso. Para isso, faremos novamente o upload dos arquivos disponibilizados (carros-aos.txt, carros-km.txt e assim por diante). No código, criaremos novamente a variável km, agora recebendo a chamada de np.loadtxt().

O primeiro parâmetro que passaremos a essa função é o fname =, que receve o nome do arquivo cujo conteúdo queremos armazenar - nesse caso, carros-km.txt. Outro parâmetro possível é o dtype, que define o tipo de dado armazenado. O padrão na criação do array é float, e usaremos a instrução dtype = int para forçarmos esse tipo.

km = np.loadtxt(fname = 'carros-km.txt', dtype = int)

kmCOPIAR CÓDIGO

A execução nos devolverá um retorno como esse:

array([ 44410, 5712, 37123, 0, 25757, 10728, 0, 77599, 99197, 37978, 12859, 8052, 89773, 0, 41457, 115607, 46449, ...])

Se quisermos, podemos verificar o tipo de dado armazenado com km.dtype.

dtype('int64')

Existem também arrays com mais de uma dimensão. Nesse caso, trabalharemos apenas com arrays de duas dimensões, deixando as outras possibilidades para cursos mais avançados. Esse tipo de array funciona como uma matriz ou um dataset. De modo a facilitarmos nosso aprendizado, usaremos a seguinte lista pré-preparada.

dados = [

['Rodas de liga', 'Travas elétricas', 'Piloto automático', 'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento', 'Sensor crepuscular', 'Sensor de chuva'],

['Central multimídia', 'Teto panorâmico', 'Freios ABS', '4 X 4', 'Painel digital', 'Piloto automático', 'Bancos de couro', 'Câmera de estacionamento'],

['Piloto automático', 'Controle de estabilidade', 'Sensor crepuscular', 'Freios ABS', 'Câmbio automático', 'Bancos de couro', 'Central multimídia', 'Vidros elétricos']

]

dadosCOPIAR CÓDIGO

Criaremos então uma variável Acessorios à qual atribuiremos a chamada de np.array(). Esta, por sua vez, receberá por parâmetro a lista dados.

Acessorios = np.array(dados)

AcessoriosCOPIAR CÓDIGO

Com isso, teremos um array numpy contendo toda a nossa lista.

array([['Rodas de liga', 'Travas elétricas', 'Piloto automático',

'Bancos de couro', 'Ar condicionado', 'Sensor de estacionamento',

'Sensor crepuscular', 'Sensor de chuva'],

['Central multimídia', 'Teto panorâmico', 'Freios ABS', '4 X 4',

'Painel digital', 'Piloto automático', 'Bancos de couro',

'Câmera de estacionamento'],

['Piloto automático', 'Controle de estabilidade',

'Sensor crepuscular', 'Freios ABS', 'Câmbio automático',

'Bancos de couro', 'Central multimídia', 'Vidros elétricos']],

dtype='<U24')COPIAR CÓDIGO

É possível verificarmos a dimensão de cada um dos arrays que acabamos de criar utilizando a instrução .shape. Começaremos com km.shape:

km.shapeCOPIAR CÓDIGO

(258, )

Isso significa que o nosso array possui 258 "linhas" e uma única dimensão. Mas observe o que acontece quando fazemos o mesmo para Acessorios:

Acessorios.shapeCOPIAR CÓDIGO

(3, 8)

Esse retorno é uma dupla indicando que o array possui **3 linhas** e **8 colunas**.

Agora que conhecemos um pouco do funcionamento desses arrays, ainda falta entendermos o motivo para utilizá-los ao invés das listas. Primeiramente, os arrays têm um desempenho muito melhor que o das listas quando fazemos operações matemáticas com eles. Para demonstrarmos isso, criaremos uma variável np\_array à qual atribuiremos a chamada de np.arange() recebendo 1000000 (um milhão) como parâmetro.

np\_array = np.arange(1000000)COPIAR CÓDIGO

Em seguida, criaremos também uma variável py\_list que usaremos para armazenar uma lista (list()) também contendo números até um milhão (range(1000000)).

py\_list = list(range(1000000))COPIAR CÓDIGO

A ideia agora é fazermos um for que multiplicará cada elemento do array np\_array por 2, e repetirá esse processo 100 vezes. Para isso, usaremos a sintaxe for \_ in range(100): np\_array \*= 2, onde o underscore (\_) indica que queremos ignorar aquela variável. Antes dessa operação, adicionaremos a instrução %time, com a qual conseguiremos verificar o tempo de execução desse código. Note que, nesse caso, conseguiremos multiplicar os valores por 2 diretamente, sem a necessidade de outro for.

%time for \_ in range(100): np\_array \*= 2COPIAR CÓDIGO

Repetiremos essa construção, dessa vez passando a lista do Python (py\_list) na operação de multiplicação. Entretanto, dessa vez será necessária a inclusão de outro for, que resumiremos com o auxilio das *list comprehensions*.

%time for \_ in range(100): py\_list = [x \* 2 for x in py\_list]COPIAR CÓDIGO

Agora compararemos os tempos de execução de cada um deles, começando pelo array numpy:

CPU times: user 65.2 ms, sys: 0 ns, total: 65.2 ms Wall time: 73.7 ms

Note que foram 73 millisegundos para terminar o processamento da operação. Já com as listas do Python:

CPU times: user 8.17 s, sys: 1.79 s, total: 9.96 s Wall time: 9.98 s

Aqui temos um tempo muito superior, de aproximadamente 10 segundos. Ou seja, a diferença de desempenho é enorme e representa uma vantagem de se utilizar o Numpy. No próximo vídeo conversaremos sobre operações aritméticas com arrays Numpy.