ARRAYS COM PYTHON + NUMPY

1ª EDIÇÃO
2020

FERNANDO FELTRIN

EBOOK


ARRAYS COM
PYTHON +
NUMPY
1ª EDIÇÃO
2020
FERNANDO FELTRIN
EBOOK

Avisos

Este livro conta com mecanismo antipirataria Amazon Kindle Protect DRM. Cada cópia possui um identificador próprio rastreável, a distribuição ilegal deste conteúdo resultará nas medidas legais cabíveis.

Sobre o Autor

Fernando Feltrin é Engenheiro da Computação com especializações na área de ciência de dados e inteligência artificial, Professor licenciado para docência de nível técnico e superior, Autor de mais de 10 livros sobre programação de computadores e responsável pelo desenvolvimento e implementação de ferramentas voltadas a modelos de redes neurais artificiais aplicadas à radiologia (diagnóstico por imagem).

Livros


Python do ZERO
à Programação Orientada a Objetos
Fernando Feltrin
FERNANDO FELTRIN
PROGRAMAÇÃO ORIENTADA A OBJETOS COM PYTHON
1ª EDIÇÃO 2020
EBOOK
Ciência de Dados e Aprendizado de Máquina
Fernando Feltrin
ARRAYS COM PYTHON + NUMPY
FERNANDO FELTRIN
FERNANDO FELTRIN
INTELIGÊNCIA ARTIFICIAL COM PYTHON
FERNANDO FELTRIN
REDES NEURAIS ARTIFICIAIS
FERNANDO FELTRIN
ANÁLISE FINANCEIRA COM PYTHON
FERNANDO FELTRIN
TÓPICOS AVANÇADOS EM PYTHON
1ª EDIÇÃO 2020
FERNANDO FELTRIN
VISÃO COMPUTACIONAL EM PYTHON
PYTHON + TENSORFLOW
Fernando Feltrin
EBOOK
PYTHON NA PRÁTICA
APRENDA PYTHON ATRAVÉS DE EXERCÍCIOS
FERNANDO FELTRIN
TÓPICOS ESPECIAIS EM PYTHON
FERNANDO FELTRIN
BLOCKCHAIN E CRIPTOMOEDAS EM PYTHON
FERNANDO FELTRIN
TÓPICOS ESPECIAIS EM PYTHON VOL. 2
FERNANDO FELTRIN
COLETÂNEA
TÓPICOS ESPECIAIS EM PYTHON VOL. 1 E 2
2 livros, 280+ páginas
FERNANDO FELTRIN

Python do ZERO à Programação Orientada a Objetos

Programação Orientada a Objetos com Python

Ciência de Dados e Aprendizado de Máquina

Inteligência Artificial com Python

Redes Neurais Artificiais com Python

Análise Financeira com Python

Arrays com Python + Numpy

Tópicos Avançados em Python

Visão Computacional em Python

Python na Prática (Exercícios resolvidos e comentados)

Tópicos Especiais em Python vol. 1

Tópicos Especiais em Python vol. 2

Blockchain e Criptomoedas em Python

Coletânea Tópicos Especiais em Python

Python na Prática (Códigos comentados)

PYTHON TOTAL (Coletânea de 12 livros)

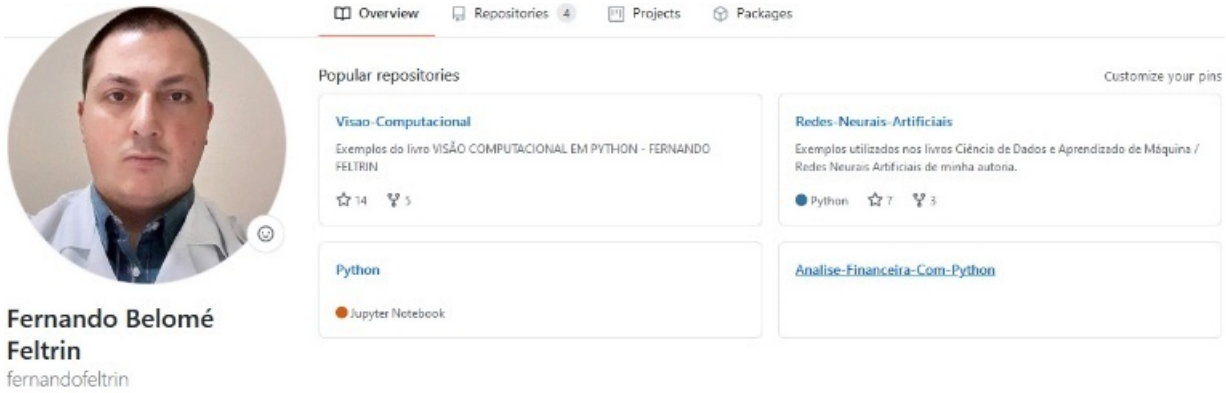
Curso

[Curso Python do ZERO à Programação Orientada a Objetos]

Mais de 15 horas de videoaulas que lhe ensinarão programação em linguagem Python de forma simples, prática e objetiva.

Redes Sociais

https://github.com/fernandofeltrin



https://github.com/fernandofeltrin

Sumário

Introdução

Quando estamos usando da linguagem Python para criar nossos códigos, é normal que usemos de uma série de recursos internos da própria linguagem, uma vez que como diz o jargão, Python vem com baterias inclusas, o que em outras palavras pode significar que Python mesmo em seu estado mais básico já nos oferece uma enorme gama de funcionalidades prontas para implementação.

No âmbito de nichos específicos como, por exemplo, computação científica, é bastante comum o uso tanto das ferramentas nativas da linguagem Python como de bibliotecas desenvolvidas pela comunidade para propósitos bastante específicos.

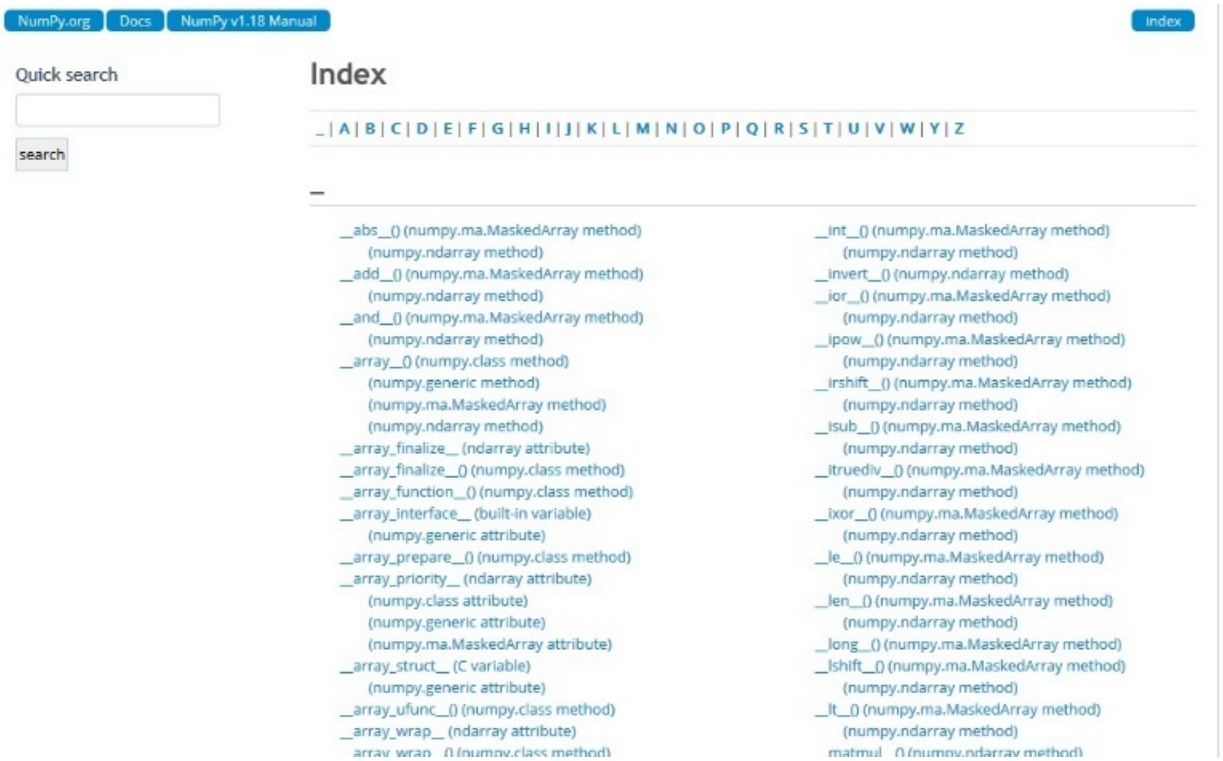
Com uma rápida pesquisa no site https://pypi.org é possível ver que existem, literalmente, milhares de bibliotecas, módulos e pacotes desenvolvidos pela própria comunidade, a fim de adicionar novas funcionalidades ou aprimorar funcionalidades já existentes no núcleo da linguagem Python.

Uma das bibliotecas mais usadas, devida a sua facilidade de implementação e uso, é a biblioteca Numpy. Esta por sua vez oferece uma grande variedade de funções aritméticas de fácil aplicação, de modo que para certos tipos de operações as funções

da biblioteca Numpy são muito mais intuitivas e eficientes do que as funções nativas de mesmo propósito presentes nas built-ins do Python.

Sobre a biblioteca Numpy

Como mencionado anteriormente, a biblioteca Numpy, uma das melhores, se não a melhor, quando falamos de bibliotecas dedicadas a operações aritméticas, possui em sua composição uma série de funções matemáticas pré-definidas, de modo que bastando importar a biblioteca em nosso código já temos à disposição as tais funções prontas para execução.



E quando digo uma enorme variedade de funções predefinidas, literalmente temos algumas centenas delas prontas para uso, bastando chamar a função específica e a parametrizar como é esperado. Por meio da documentação online da biblioteca é possível ver a lista de todas as funções.

Neste pequeno livro, iremos fazer uma abordagem simples das principais funcionalidades que a biblioteca Numpy nos oferece, haja visto que a aplicação de tais funções se dá de acordo com a necessidade, e em suas rotinas você notará que a grande maioria das funções disponíveis nem mesmo é usada, porém, todas elas estão sempre à disposição prontas para uso.

Também será visto nos capítulos finais um exemplo de funções da biblioteca Numpy sendo aplicadas em machine learning.

Instalação e Importação das Dependências

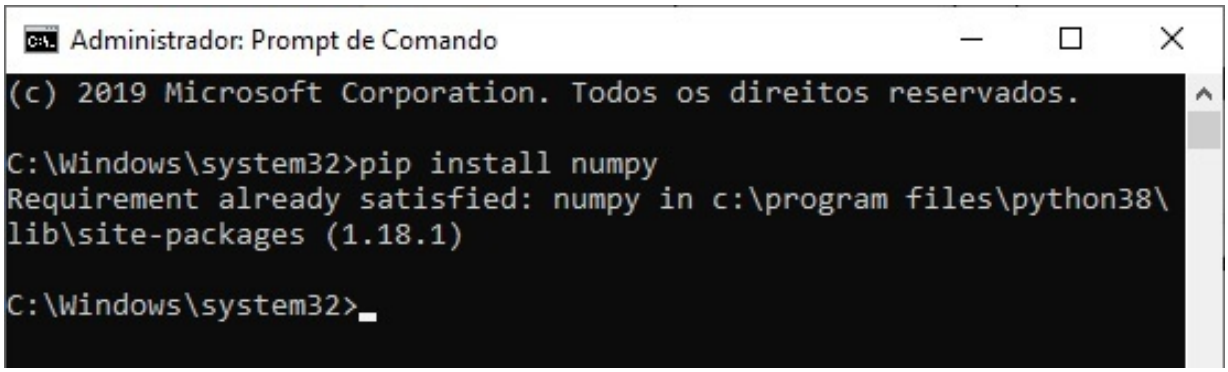
Usando do Google Colab, podemos realizar a instalação e atualização da biblioteca Numpy por meio do gerenciador de pacotes

pip, de forma muito parecida como quando instalamos qualquer outra biblioteca localmente.

```
! pip install numpy
```

Uma vez que o processo de instalação tenha sido finalizado como esperado, sem erros, podemos imediatamente fazer o uso das ferramentas disponíveis da biblioteca Numpy.

O processo de instalação local também pode ser feito via Prompt de Comando, por meio de pip install numpy.



Para darmos início a nossos exemplos, já com nossa IDE aberta, primeiramente é necessário realizar a importação da biblioteca Numpy, uma vez que a mesma é uma biblioteca externa que por padrão não vem pré-carregada em nossa IDE.

O processo de importação é bastante simples, bastando criar a linha de código referente a importação da biblioteca logo no início de nosso código para que a biblioteca seja carregada em prioritariamente antes das demais estruturas de código, de acordo com a leitura léxica de nosso interpretador.

```
import numpy as np
```

Uma prática comum é referenciar nossas bibliotecas por meio de alguma sigla, para que simplesmente fique mais fácil instanciar a mesma em nosso código. Neste caso, importamos a biblioteca numpy e a referenciamos como np apenas por convenção.

Trabalhando com Arrays

Criando uma array numpy

Estando todas as dependências devidamente instaladas e carregadas, podemos finalmente dar início ao entendimento das estruturas de dados da biblioteca Numpy. Basicamente, já que estaremos trabalhando com dados por meio de uma biblioteca

científica, é necessário contextualizarmos os tipos de dados envolvidos assim como as possíveis aplicações dos mesmos.

Sempre que estamos trabalhando com dados por meio da biblioteca Numpy, a primeira coisa que devemos ter em mente é que todo e qualquer tipo de dado em sua forma mais básica estará no formato de uma Array Numpy.

Quando estamos falando de arrays de forma geral basicamente estamos falando de dados em formato de vetores / matrizes, representados normalmente com dimensões bem definidas, alocando dados nos moldes de uma tabela com suas respectivas linhas, colunas e camadas.

A partir do momento que estamos falando de uma array "do tipo numpy", basicamente estamos falando que tais dados estão carregados pela biblioteca Numpy de modo que possuem um sistema de indexação próprio da biblioteca e algumas métricas de mapeamento dos dados para que se possam ser aplicados sobre os mesmos uma série de funções, operações lógicas e aritméticas dependendo a necessidade.

```
data = np.array ([ 2 ,  4 ,  6 ,  8 ,  10 ])
```

Inicialmente criamos uma variável / objeto de nome data, que recebe como atributo a função np.array( ) parametrizada com uma simples lista de caracteres. Note que instanciamos a biblioteca numpy por meio de np, em seguida chamando a função np.array( ) que serve para transformar qualquer tipo de dado em array do tipo numpy.

```
print ( data )
print ( type ( data ))
print ( data.shape )
```

Uma vez criada a array data com seus respectivos dados/valores, podemos realizar algumas verificações simples.

```
[ 2  4  6  8 10]
<class 'numpy.ndarray'>
(5,)
```

Na primeira linha vemos o retorno da função print( ) simplesmente exibindo o conteúdo de data. Na segunda linha o retorno referente a função print( ) parametrizada para verificação do tipo de dado por meio da função type( ) por sua vez parametrizado com data, note que, como esperado, trata-se de uma 'numpy.ndarray'.

Por fim, na terceira linha temos o retorno referente ao tamanho da array por meio do método .shape, nesse caso, 5 elementos em uma linha, o espaço vazio após a vírgula indica não haver nenhuma coluna adicional.

Criando uma array gerada com números ordenados

Dependendo o propósito, pode ser necessário a criação de uma array composta de números gerados ordenadamente dentro de um intervalo numérico. Nesse contexto, a função np.arange( ) foi criada para este fim, bastando especificar o número de elementos que irão compor a array.

```
data = np.arange ( 15 )

print ( data )
```

Nesse caso, quando estamos falando em números ordenados estamos falando em números inteiros gerados de forma sequencial, do zero em diante.

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

Novamente por meio da função print( ) parametrizada com data, é possível ver o retorno, nesse caso, uma array sequencial de 15 elementos gerados de forma ordenada.

Array gerada com números do tipo float

```
data = np.random.rand ( 15 )

print ( data )
```

Outra possibilidade que temos é a de criar uma array numpy de dados aleatórios tipo float, escalonados entre 0 e 1, por meio da função np.random.rand( ), novamente parametrizada com o número de elementos a serem gerados.

```
[0.18087199 0.32988976 0.73241758 0.5836709  0.02009198 0.42970031
 0.0398033  0.28778931 0.29019795 0.29858361 0.30491469 0.68747359
 0.99062792 0.71303427 0.16030855]
```

Da mesma forma, por meio da função print( ) é possível visualizar tais números gerados randomicamente. Note que neste caso os números são totalmente aleatórios, não sequenciais, dentro do intervalo de 0 a 1.

Array gerada com números do tipo int

```
data = np.random.randint ( 10 ,  size =  10 )

print ( data )
```

Outro meio de criar uma array numpy é gerando números inteiros dentro de um intervalo específico definido manualmente para a

função np.random.randint( ). Repare que o primeiro parâmetro da função, nesse caso, é o número 10, o que em outras palavras nos diz que estamos gerando números de 0 até 10, já o segundo parâmetro, size, aqui definido em 10, estipula que 10 elementos serão gerados aleatoriamente.

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

Mais uma vez por meio da função print( ) podemos visualizar a array gerada, como esperado, 10 números inteiros gerados aleatoriamente.

```
data = np.random.random (( 2 ,  2 ))

print ( data )
```

De forma parecida com o que fizemos anteriormente por meio da função np.random.rand( ), podemos via np.random.random( ) gerar arrays de dados tipo float, no intervalo entre 0 e 1, aqui com mais de uma dimensão. Note que para isso é necessário colocar o número de linhas e colunas dentro de um segundo par de parênteses.

```
[[0.10310535 0.87117662]
 [0.4818465  0.4363957 ]]
```

Como de costume, por meio da função print( ) visualizamos os dados gerados e atribuídos a data.

Array gerada com números zero

```
data = np.zeros (( 3 ,  4 ))

print ( data )
```

Dependendo da aplicação, pode ser necessário gerar arrays numpy previamente preenchidas com um tipo de dado específico. Em machine learning, por exemplo, existem várias aplicações onde se cria uma matriz inicialmente com valores zerados a serem substituídos após algumas funções serem executadas e gerarem retornos.

Aqui, por meio da função np.zeros( ) podemos criar uma array com as dimensões que quisermos, totalmente preenchida com números 0. Nesse caso, uma array com 3 linhas e 4 colunas deverá ser gerada.

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Por meio da função print( ) novamente vemos o resultado, neste caso, uma matriz composta apenas de números 0.

Array gerada com números um

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

print ( data )
```

Da mesma forma como realizado no exemplo anterior, de acordo com as particularidades de uma aplicação pode ser necessário realizar a criação de uma array numpy inicialmente preenchida com números 1. Apenas substituindo a função np.zeros( ) por np.ones( ) criamos uma array nos mesmos moldes da anterior.

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

Por meio da função print( ) visualizamos o resultado, uma matriz de números 1.

Array gerada com espaços vazios

```
data = np.empty (( 3 ,  4 ))

print ( data )
```

Apenas como curiosidade, teoricamente é possível criar uma array numpy de dimensões predefinidas e com valores vazios por meio da função np.empty( ).

```
[[4.9e-323 1.5e-323 9.9e-324 1.5e-323]
 [2.0e-323 3.0e-323 2.0e-323 0.0e+000]
 [0.0e+000 9.9e-324 2.0e-323 0.0e+000]]
```

Porém o retorno gerado, como você pode visualizar na imagem acima, é uma array com números em notação científica, representando algo muito próximo de zero, porém não zero absoluto.

```
data = np.arange ( 10 )  *  -1

print ( data )
```

Concluindo essa linha de raciocínio, novamente dependendo da aplicação, pode ser necessário a geração de uma array composta de números negativos. Nesse caso não há uma função específica para tal fim, porém há o suporte da linguagem Python em permitir o uso do operador de multiplicação por um número negativo.

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

Por meio da função print( ), como esperado, dessa vez temos uma array composta de 10 elementos negativos.

Criando uma array de números aleatórios, mas com tamanho predefinido em variável

Uma última possibilidade a ser explorada é a de associar a criação de uma array com um tamanho predefinido através de uma variável. Por meio da função np.random.permutation( ) podemos definir um tamanho que pode ser modificado conforme a necessidade,

```
tamanho =  10

data6 = np.random.permutation ( tamanho )

print ( data6 )
```

Para esse fim, inicialmente criamos uma variável de nome tamanho que recebe como atributo o valor 10, este será o número a ser usado como referência de tamanho para a array, em outras palavras, o número de elementos da mesma.

Na sequência criamos nossa array data6 que recebe a função np.random.permitation( ) por sua vez parametrizada com o valor de tamanho. Por meio da função print( ) podemos ver o resultado desse gerador.

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

O resultado, como esperado, é uma array unidimensional composta por 10 elementos aleatoriamente gerados.

Criando arrays de dimensões específicas

Uma vez que estamos trabalhando com dados em forma matricial (convertidos e indexados como arrays numpy), uma característica importante a observar sobre os mesmos é sua forma representada, seu número de dimensões.

Posteriormente estaremos entendendo como realizar diversas operações entre arrays, mas por hora, raciocine que para que seja possível o cruzamento de dados entre arrays, as mesmas devem possuir formato compatível, nessa lógica, posteriormente veremos que para determinadas situações estaremos inclusive realizando a alteração do formato de nossas arrays para aplicação de certas funções.

Por hora, vamos entender da maneira correta como são definidas as dimensões de uma array numpy.

Array unidimensional

```
data = np.random.randint ( 5 ,  size =  10

print ( data )
print ( data.shape )
```

Como já feito anteriormente, por meio da função np.random.randint( ) podemos gerar uma array, nesse caso, de 10 elementos com valores distribuídos entre 0 a 5.

```
[4 4 2 0 4 4 3 0 2 1]
(10,)
```

Por meio da função print( ) parametrizada com data vemos os valores da array, e parametrizando a mesma com data.shape podemos inspecionar de forma simples o formato de nossa array. Neste caso, [10,] representa uma array com 10 elementos em apenas uma linha. Uma array unidimensional.

Array bidimensional

```
data2 = np.random.randint ( 5 ,  size =  ( 3 ,  4 ))

print ( data2 )
print ( data2.shape )
```

Para o próximo exemplo criamos uma variável de nome data2 que recebe, da mesma forma que o exemplo anterior, a função np.random.randint( ) atribuída para si, agora substituindo o valor de size por dois valores (3, 4), estamos definindo manualmente que o tamanho dessa array deverá ser de 3 linhas e 4 colunas, respectivamente. Uma array bidimensional.

```
[[3 1 1 4]
 [3 2 1 2]
 [3 1 3 0]]
(3, 4)
```

Como esperado, por meio da função print( ) vemos a array em si com seu conteúdo, novamente via data2.shape vemos o formato esperado.

Array tridimensional

```
data3 = np.random.randint ( 5 ,  size =  ( 5 ,  3 ,  4 ))

print ( data3 )
print ( data3.shape )
```

Seguindo com a mesma lógica, criamos a variável data3 que chama a função np.random.randint( ), alterando novamente o número

definido para o parâmetro size, agora com 3 parâmetros, estamos definindo uma array de formato tridimensional.

Nesse caso, importante entender que o primeiro dos três parâmetros se refere ao número de camadas que a array terá em sua terceira dimensão, enquanto o segundo parâmetro define o número de linhas e o terceiro parâmetro o número de colunas.

```
[[[4 0 2 1]
  [4 1 3 4]
  [0 1 4 4]]

 [[1 0 1 0]
  [1 0 4 3]
  [3 1 1 1]]

 [[1 0 2 2]
  [3 2 2 2]
  [2 3 3 4]]

 [[1 4 1 4]
  [1 4 2 3]
  [1 2 3 2]]

 [[0 3 1 0]
  [4 0 0 4]
  [4 2 4 4]]]
(5, 3, 4)
```

Como esperado, 5 camadas, cada uma com 3 linhas e 4 colunas, preenchida com números gerados aleatoriamente entre 0 e 5.

```
data4 = np.array ([[ 1 ,  2 ,  3 ,  4 ],  [ 1 ,  3 ,  5 ,  7 ]])

print ( data4 )
```

Entendidos os meios de como gerar arrays com valores aleatórios, podemos agora entender também que é perfeitamente possível criar arrays manualmente, usando de dados em formato de lista para a construção da mesma. Repare que seguindo uma lógica parecida

com o que já foi visto anteriormente, aqui temos duas listas de números, consequentemente, teremos uma array bidimensional.

```
[[1 2 3 4]
 [1 3 5 7]]
```

Exibindo em tela o resultado via função print( ) temos nossa array com os respectivos dados declarados manualmente. Uma última observação a se fazer é que imprescindível que se faça o uso dos dados declarados da maneira correta, inclusive na sequência certa.

Verificando o tamanho e formato de uma array

Quando estamos trabalhando com dados em formato de vetor ou matriz, é muito importante eventualmente verificar o tamanho e formato dos mesmos, até porquê para ser possível realizar operações entre tais tipos de dados os mesmos devem ter um formato padronizado.

```
d ata3 = np.random.randint ( 5 ,  size =  ( 5 ,  3 ,  4 ))

print ( data3.ndim )
print ( data3.shape )
print ( data3.size )
```

Indo diretamente ao ponto, nossa variável data3 é uma array do tipo numpy como pode ser observado. A partir daí é possível realizar algumas verificações, sendo a primeira delas o método .ndim, que por sua vez retornará o número de dimensões presentes em nossa array. Da mesma forma .shape nos retorna o formato de nossa array (formato de cada dimensão da mesma) e por fim .size retornará o tamanho total dessa matriz.

```
3
(5, 3, 4)
60
```

Como retorno das funções print( ) criadas, logo na primeira linha temos o valor 3 referente a .ndim, que em outras palavras significa tratar-se de uma array tridimensional. Em seguida, na segunda linha temos os valores 5, 3, 4, ou seja, 5 camadas, cada uma com 3 linhas e 4 colunas. Por fim na terceira linha temos o valor 60, que se refere a soma de todos os elementos que compõe essa array / matriz.

Verificando o tamanho em bytes de um item e de toda a array

Uma das questões mais honestas que todo estudante de Python (não somente Python, mas outras linguagens) possui é do por quê utilizar de bibliotecas externas quando já temos a disposição ferramentas nativas da linguagem.

Pois bem, toda linguagem de programação de código fonte aberto tende a receber forte contribuição da própria comunidade de usuários, que por sua vez, tentam implementar novos recursos ou aperfeiçoar os já existentes.

Com Python não é diferente, existe uma infinidade de bibliotecas para todo e qualquer tipo de aplicação, umas inclusive, como no caso da Numpy, mesclam implementar novas ferramentas assim como reestruturar ferramentas padrão em busca de maior performance.

Uma boa prática é manter em nosso código apenas o necessário, assim como levar em consideração o impacto de cada elemento no processamento de nossos códigos. Não é porque você vai criar um editor de texto que você precise compilar junto todo o sistema operacional. Inclusive se você já tem alguma familiaridade com a programação em Python sabe muito bem que sempre que possível importamos apenas os módulos e pacotes necessários de uma biblioteca, descartando todo o resto.

```
print ( f 'Cada elemento possui  { data3.itemsize }  bytes' )
```

```
print ( f 'Toda a matriz possui  { data3.nbytes }  bytes' )
```

Dado o contexto acima, uma das verificações comumente realizada é a de consultar o tamanho de cada elemento que faz parte da array assim como o tamanho de toda a array. Isso é feito diretamente por meio dos métodos .itemsize e .nbytes, respectivamente, Aqui, apenas para variar um pouco, já implementamos essas verificações diretamente como parâmetro em nossa função print( ), porém é perfeitamente possível associar esses dados a uma variável.

```
Cada elemento possui 8 bytes
Toda a matriz possui 480 bytes
```

Como esperado, junto de nossa mensagem declarada manualmente por meio de f'Strings temos os respectivos dados de cada elemento e de toda a matriz

Verificando o elemento de maior valor de uma array

Iniciando o entendimento das possíveis verificações que podemos realizar em nossas arrays, uma das mais comuns é buscar o elemento de maior valor em relação aos demais da array.

```
data = np.arange ( 15 )

print ( data )
print ( data. max ())
```

Criada a array data com 15 elementos ordenados, por meio da função print( ) parametrizando a mesma com os dados de data veremos tais elementos. Aproveitando o contexto, por meio da função max( ) teremos acesso ao elemento de maior valor associado.

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

Como esperado, na primeira linha temos todos os elementos de data, na segunda linha temos em destaque o número 14, lembrando que a array tem 15 elementos, mas os mesmos iniciam em 0, logo, temos números de 0 a 14, sendo 14 o maior deles.

Consultando um elemento por meio de seu índice

Como visto anteriormente, a organização e indexação dos elementos de uma array se assemelha muito com a estrutura de dados de uma lista, e como qualquer lista, é possível consultar um determinado elemento por meio de seu índice, mesmo que esse índice não seja explícito.

Inicialmente vamos ver alguns exemplos com base em array unidimensional.

```
data = np.arange ( 15 )

print ( data )
print ( data [ 8 ])
```

Criada a array data da mesma forma que o exemplo anterior, por meio da função print( ) parametrizada com data[ ] especificando um número de índice, é esperado que seja retornado o respectivo dado/valor situado naquela posição.

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

Se não houver nenhum erro de sintaxe, é retornado o dado/valor da posição 8 do índice de nossa array data. Nesse caso, o retorno é o próprio número 8 uma vez que temos dados ordenados nesta array.

```
data
```

```
print ( data )
print ( data [ -5 ])
```

Da mesma forma, passando como número de índice um valor negativo, será exibido o respectivo elemento a contar do final para o começo dessa array.

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

Nesse caso, o 5º elemento a contar do fim para o começo é o número 10.

Partindo para uma array multidimensional, o processo para consultar um elemento é feito da mesma forma dos exemplos anteriores, bastando agora especificar a posição do elemento levando em conta a linha e coluna onde é esperado.

```
data2 = np.random.randint ( 5 ,  size =  ( 3 ,  4 ))

print ( data2 )
print ( data2 [ 0 ,  2 ])
```

Note que é criada a array data2, de elementos aleatórios entre 0 e 5, tendo 3 linhas e 4 colunas em sua forma. Parametrizando print com data2[0, 2] estamos pedindo que seja exibido o elemento situado na linha 0 (primeira linha) e coluna 2 (terceira coluna pois a primeira coluna é 0).

```
[[0 1 4 2]
 [4 3 4 0]
 [1 0 1 1]]
4
```

Podemos visualizar via console a array em si, e o respectivo elemento situado na primeira linha, terceira coluna. Neste caso, o número 4.

Consultando elementos dentro de um intervalo

```
data

print ( data )
```

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

Reutilizando nossa array data criada anteriormente, uma simples array unidimensional de 15 elementos ordenados, podemos avançar com o entendimento de outras formas de consultar dados/valores, dessa vez, definindo intervalos específicos.

```
print ( data )
print ( data [: 5 ])
print ( data [ 3 :])
print ( data [ 4 : 8 ])
print ( data [:: 2 ])
print ( data [ 3 ::])
```

Como mencionado anteriormente, a forma como podemos consultar elementos via índice é muito parecida (senão igual) a forma como realizávamos as devidas consultas em elementos de uma lista. Sendo assim, podemos usar das mesmas notações aqui.

Na primeira linha simplesmente por meio da função print( ) estamos exibindo em tela os dados contidos em data.

Na sequência, passando data[:5] (mesmo que data[0:5]) como parâmetro para nossa função print( ) estaremos exibindo os 5 primeiros elementos dessa array. Em seguida via data[3:] estaremos

exibindo do terceiro elemento em diante todos os demais. Na sequência, via data[4:8] estaremos exibindo do quarto ao oitavo elemento.

Da mesma forma, porém com uma pequena diferença de notação, podemos por meio de data[::2] exibir de dois em dois elementos, todos os elementos. Por fim, via data[3::] estamos pulando os 3 primeiros elementos e exibindo todos os demais.

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

Como esperado, na primeira linha temos todos os dados da array, na segunda linha apenas os 5 primeiros, na terceira linha do terceiro elemento em diante, na quarta linha os elementos entre 4 e 8, na quinta linha os elementos pares e na sexta linha todos os dados exceto os três primeiros.

Modificando manualmente um dado/valor de um elemento por meio de seu índice.

Assim como em listas podíamos realizar modificações diretamente sobre seus elementos, bastando saber seu índice, aqui estaremos realizando o mesmo processo da mesma forma.

```
data2

print ( data2 )

data2 [ 0 ,  0 ] = 10

print ( data2 )
```

Reutilizando nossa array data 2 criada anteriormente, podemos realizar a alteração de qualquer dado/valor desde que se faça a referência correta a sua posição na array de acordo com seu índice. Nesse caso, apenas como exemplo, iremos substituir o valor do elemento situado na linha 0 e coluna 0 por 10.

```
[[1 3 2 3]
 [4 0 4 0]
 [0 2 4 0]]
[[10  3  2  3]
 [ 4  0  4  0]
 [ 0  2  4  0]]
```

Repare que como esperado, o valor do elemento [0, 0] foi substituído de 1 para 10 como mostra o retorno gerado via print( ).

```
data2

data2 [ 1 ,  1 ] = 6.82945

print ( data2 )
```

Apenas como curiosidade, atualizando o valor de um elemento com um número do tipo float, o mesmo será convertido para int para que não hajam erros por parte do interpretador.

```
[[10  3  2  3]
 [ 4  6  4  0]
 [ 0  2  4  0]]
```

Elemento [1, 1] em formato int, sem as casas decimais declaradas manualmente.

```
data2 = np.array ([ 8 ,  -3 ,  5 ,  9 ], dtype =  'float' )

print ( data2 )
```

```
print ( type ( data2 [ 0 ]))
```

Supondo que você realmente precise dos dados em formato float dentro de uma array, você pode definir manualmente o tipo de dado por meio do parâmetro dtype = 'float'. Dessa forma, todos os elementos desta array serão convertidos para float.

Aqui como exemplo estamos criando novamente uma array unidimensional, com números inteiros em sua composição no momento da declaração.

```
[ 8. -3.  5.  9.]
<class 'numpy.float64'>
```

Analisando o retorno de nossa função print( ) podemos ver que de fato são números, agora com casas decimais, de tipo float64.

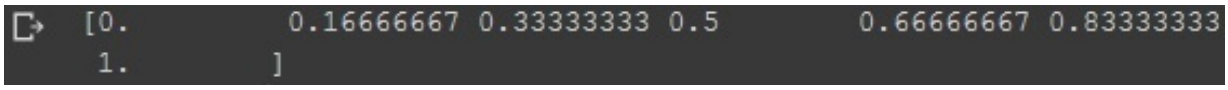
Criando uma array com números igualmente distribuídos

Dependendo mais uma vez do contexto, pode ser necessária a criação de uma array com dados/valores distribuídos de maneira uniforma na mesma. Tenha em mente que os dados gerados de forma aleatória não possuem critérios definidos quanto sua organização. Porém, por meio da função linspace( ) podemos criar dados uniformemente arranjados dentro de uma array.

```
data = np.linspace ( 0 ,  1 ,  7 )

print ( data )
```

Criamos uma nova array de nome data, onde por meio da função np.linspace( ) estamos gerando uma array composta de 7 elementos igualmente distribuídos, com valores entre 0 e 1.



O retorno como esperado é uma array de números float, para que haja divisão igual dos valores dos elementos.

Redefinindo o formato de uma array

Um dos recursos mais importantes que a biblioteca Numpy nos oferece é a de poder livremente alterar o formato de nossas arrays. Não que isso não seja possível sem o auxílio da biblioteca Numpy, mas a questão mais importante aqui é que no processo de reformatação de uma array numpy, a mesma se reajustará em todos os seus níveis, assim como atualizará seus índices para que não se percam dados entre dimensões.

```
data5 = np.arange ( 8 )

print ( data5 )
```

Inicialmente apenas para o exemplo criamos nossa array data5 com 8 elementos gerados e ordenados por meio da função np.arange( ).

```
data5 = data5.reshape ( 2 ,  4 )

print ( data5 )
```

Em seguida, por meio da função reshape( ) podemos alterar livremente o formato de nossa array. Repare que inicialmente geramos uma array unidimensional de 8 elementos, agora estamos transformando a mesma para uma array bidimensional de 8 elementos, onde os mesmos serão distribuídos em duas linhas e 4 colunas.

Importante salientar que você pode alterar livremente o formato de uma array desde que mantenha sua proporção de distribuição de dados. Aqui, nesse exemplo bastante simples, 8 elementos dispostos em uma linha passarão a ser 8 elementos dispostos em duas linhas e 4 colunas. O número de elementos em si não pode ser alterado.

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

Na primeira linha, referente ao primeiro print( ) temos a array data5 em sua forma inicial, na segunda linha o retorno obtido pós reshape.

Usando de operadores lógicos em arrays

Dependendo o contexto pode ser necessário fazer uso de operadores lógicos mesmo quando trabalhando com arrays numpy. Respeitando a sintaxe Python, podemos fazer uso de operadores lógicos sobre arrays da mesma forma como fazemos com qualquer outro tipo de dado.

```
data7 = np.arange ( 10 )

print ( data7 )
print ( data7 >  3 )
```

Para esse exemplo criamos nossa array data7 de 10 elementos ordenados gerados aleatoriamente. Por meio da função print( ) podemos tanto exibir seu conteúdo quanto realizar proposições baseadas em operadores lógicos. Note que no exemplo a seguir, estamos verificando quais elementos de data7 tem seu valor maior que 3.

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

Na primeira linha todo o conteúdo de data7, na segunda marcados como False os elementos menores que 3, assim como marcados como True os de valor maior que 3.

```
print ( data7 )
print ( data7 [ 4 ]  >  5 )
```

Da mesma forma como visto anteriormente, fazendo a seleção de um elemento por meio de seu índice, também podemos aplicar um operador lógico sobre o mesmo.

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

Nesse caso, o elemento da posição 4 do índice, o número 3, não é maior que 5, logo, o retorno sobre essa operação lógica é False.

Usando de operadores aritméticos em arrays

Dentro das possibilidades de manipulação de dados em arrays do tipo numpy está a de realizar, como esperado, operações aritméticas entre tais dados. Porém recapitulando o básico da linguagem Python, quando estamos trabalhando com determinados tipos de dados temos de observar seu tipo e compatibilidade com outros tipos de dados, sendo em determinados casos necessário a realização da conversão entre tipos de dados.

Uma array numpy permite realizar determinadas operações que nos mesmos tipos de dados em formato não array numpy iriam gerar exceções ou erros de interpretação.

```
data8 = [ 1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ]

print ( data8 +  10 )
```

Para entendermos melhor, vamos usar do seguinte exemplo, temos uma array data8 com uma série de elementos dispostos em formato de lista, o que pode passar despercebido aos olhos do usuário, subentendendo que para este tipo de dado é permitido realizar qualquer tipo de operação. Por meio da função print( ) parametrizada com data8 + 10, diferentemente do esperado, irá gerar um erro.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-61e14f649c93> in <module>()
      1 data8 = [1, 2, 3, 4, 5, 6, 7, 8]
----> 2 print(data8 + 10)

TypeError: can only concatenate list (not "int") to list
```

Repare que o interpretador lê os dados de data8 como uma simples lista, tentando por meio do operador + concatenar os dados com o valor 10 ao invés de realizar a soma.

```
data8 = np.array ( data8 )

print ( data8 +  10 )
```

Realizando a conversão de data8 para uma array do tipo numpy, por meio da função np.array( ) parametrizada com a própria variável data8, agora é possível realizar a operação de soma dos dados de data8 com o valor definido 10.

```
[1, 2, 3, 4, 5, 6, 7, 8]
[11 12 13 14 15 16 17 18]
```

Como retorno da função print( ), na primeira linha temos os valores iniciais de data8, na segunda linha os valores após somar cada elemento ao número 10.

Criando uma matriz diagonal

Aqui mais um dos exemplos de propósito específico, em algumas aplicações de machine learning é necessário a criação de uma matriz diagonal, e a mesma pode ser feita por meio da função np.diag( ).

```
data9 = np.diag (( 1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ))

print ( data9 )
```

Para esse exemplo criamos a array data9 que recebe como atributo uma matriz diagonal de 8 elementos definidos manualmente, gerada pela função np.diag( ).

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

O retorno é uma matriz gerada com os elementos predefinidos compondo uma linha vertical, sendo todos os outros espaços preenchidos com 0.

Algo parecido pode ser feito através da função np.eye( ), porém, nesse caso será gerada uma matriz diagonal com valores 0 e 1, de tamanho definido pelo parâmetro repassado para função.

```
data9 = np.eye ( 4 )

print ( data9 )
```

Neste caso, para a variável data9 está sendo criada uma array diagonal de 4 linhas e colunas conforme a parametrização realizada em np.eye( ).

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

O retorno é uma matriz diagonal, com números 1 dispostos na coluna diagonal

Criando padrões duplicados

Por meio da simples notação de "encapsular" uma array em um novo par de colchetes e aplicar a função np.tile( ) é possível realizar a duplicação/replicação da mesma quantas vezes for necessário.

```
data10 = np.tile ( np.array ([[ 9 ,  4 ],  [ 3 ,  7 ]]),  4 )

print ( data10 )
```

Aqui criada a array data10, note que atribuído para a mesma está a função np.tile( ) parametrizada com uma array definida manualmente assim como um último parâmetro 4, referente ao número de vezes a serem replicados os dados/valores da array.

```
[[9 4 9 4 9 4 9 4]
 [3 7 3 7 3 7 3 7]]
```

No retorno podemos observar os elementos declarados na array, em sua primeira linha 9 e 4, na segunda, 3 e 7, repetidos 4 vezes.

```
data10 = np.tile ( np.array ([[ 9 ,  4 ],  [ 3 ,  7 ]]),  ( 2 , 2 ))

print ( data10 )
```

Mesmo exemplo anterior, agora parametrizado para replicação em linhas e colunas.

```
[[9 4 9 4]
 [3 7 3 7]
 [9 4 9 4]
 [3 7 3 7]]
```

Note que, neste caso, a duplicação realizada de acordo com a parametrização ocorre sem sobrepor a ordem original dos elementos.

Somando um valor a cada elemento da array

Como visto anteriormente, a partir do momento que uma matriz passa a ser uma array do tipo numpy, pode-se perfeitamente aplicar sobre a mesma qualquer tipo de operador lógico ou aritmético. Dessa forma, é possível realizar tais operações, desde que leve em consideração que a operação realizada se aplicará a cada um dos elementos da array.

```
data11 = np.arange ( 0 ,  15 )

print ( data11 )

data11 = np.arange ( 0 ,  15 ) + 1

print ( data11 )
```

Para esse exemplo criamos uma nova array de nome data11, por sua vez gerada com números ordenados. Em seguida é realizada uma simples operação de somar 1 a própria array, e como dito anteriormente, essa soma se aplicará a todos os elementos.

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

Observando os retornos das funções print( ) na primeira linha temos a primeira array com seus respectivos elementos, na segunda, a mesma array onde a cada elemento foi somado 1 ao seu valor original.

```
data11 = np.arange ( 0 ,  30 , 3 ) + 3

print ( data11 )
```

Apenas como exemplo, usando da notação vista anteriormente é possível realizar a soma de um valor a cada elemento de uma array, nesse caso, somando 3 ao valor de cada elemento, de 3 em 3 elementos de acordo com o parâmetro passado para função np.arange( ).

```
[ 3  6  9 12 15 18 21 24 27 30]
```

Como esperado, o retorno é uma array gerada com números de 0 a 30, com intervalo de 3 entre cada elemento.

Realizando soma de arrays

Entendido boa parte do processo lógico das possíveis operações em arrays não podemos nos esquecer que é permitido também a soma de arrays, desde que as mesmas tenham o mesmo formato ou número de elementos.

```
d1 = np.array ([  3 ,  6 ,  9 ,  12 ,  15 ])
d2 = np.array ([  1 ,  2 ,  3 ,  4 ,  5 ])

d3 = d1 + d2

print ( d3 )
```

Para esse exemplo criamos duas arrays d1 e d2, respectivamente, cada uma com 5 elementos distintos. Por meio de uma nova variável de nome e3 realizamos a soma simples entre d1 e d2, e como esperado, cada elemento de cada array será somado ao seu elemento equivalente da outra array.

```
[ 4  8 12 16 20]
```

O resultado obtido é a simples soma do primeiro elemento da primeira array com o primeiro elemento da segunda array, assim como todos os outros elementos com seu equivalente.

```
d1 = np.array ([  3 ,  6 ,  9 ,  12 ,  15 ])
d2 = np.array ([  1 ,  2 ,  3 ,  4 ,  5 ])

d3 = d1 - d2

print ( d3 )
```

Subtração entre arrays

Exatamente da mesma forma é possível realizar a subtração entre arrays.

```
[ 2  4  6  8 10]
```

Como esperado, é obtido os valores das subtrações entre cada elemento da array d1 com o seu respectivo elemento da array d2.

Multiplicação e divisão entre arrays

```
d1 = np.array ([  3 ,  6 ,  9 ,  12 ,  15 ])
d2 = np.array ([  1 ,  2 ,  3 ,  4 ,  5 ])
```

```
d3 = d1 / d2

d4 = d1 * d2

print ( d3 )
print ( d4 )
```

Exatamente da mesma forma é possível realizar a divisão e a multiplicação entre os elementos das arrays.

```
[3. 3. 3. 3. 3.]
[ 3 12 27 48 75]
```

Obtendo assim, na primeira linha o retorno da divisão entre os elementos das arrays, na segunda linha o retorno da multiplicação entre os elementos das mesmas.

Realizando operações lógicas entre arrays

Da mesma forma que é possível realizar o uso de operadores aritméticos para seus devidos cálculos entre elementos de arrays, também é uma prática perfeitamente possível fazer o uso de operadores lógicos para verificar a equivalência de elementos de duas ou mais arrays.

```
d1 = np.array ([  3 ,   6 ,   9 ,  12 ,  15 ,  18 ,  21 ])
d2 = np.array ([  1 ,   2 ,   3 ,   4 ,   5 ,  20 ,  25 ])

d3 = d2 > d1

print ( d3 )
```

Neste exemplo, duas arrays d1 e d2 novamente, agora com algumas pequenas mudanças em alguns elementos, apenas para tornar nosso exemplo mais interessante. Em nossa variável d3 estamos usando um operador para verificar se os elementos da array d2 são maiores que os elementos equivalentes em d1. Novamente, o retorno será gerado mostrando o resultado dessa operação lógica para cada elemento.

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

Repare que por exemplo o 4º elemento da array d2 se comparado com o 4º elemento da array d1 retornou False, pois 4 não é maior que 12. Da mesma forma o último elemento de d2 se comparado com o último elemento de d1 retornou True, pois obviamente 25 é maior que 22.

Transposição de arrays

Em determinadas aplicações, principalmente em machine learning, é realizada a chamada transposição de matrizes, que nada mais é do que realizar uma alteração de forma da mesma, transformando linhas em colunas e vice versa por meio da função transpose( ).

```
arr1 = np.array ([[ 1 ,  2 ,  3 ],  [ 4 ,  5 ,  6 ]])

print ( arr1 )
print ( arr1.shape )

arr1_transposta = arr1.transpose ()

print ( arr1_transposta )
print ( arr1_transposta.shape )
```

Note que inicialmente é criada uma array de nome arr1, bidimensional, com elementos distribuídos em 2 linhas e 3 colunas.

Na sequência é criara uma nova variável de nome arr1_transposta que recebe o conteúdo de arr1, sob a função transpose( ). Como sempre, por meio da função print( ) realizaremos as devidas verificações.

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

Como esperado, na primeira parte de nosso retorno temos a array arr1 em sua forma original, com shape (2, 3) e logo abaixo a array arr1_transposta, que nada mais é do que o conteúdo de arr1 reorganizados no que diz respeito as suas linhas e colunas, nesse caso com o shape (3, 2).

Salvando uma array no disco local

```
data = np.array ([  3 ,   6 ,   9 ,  12 ,  15 ])
```

```
np.save ( 'minha_array' ,  data )
```

Encerrando nossos estudos, vamos ver como é possível salvar nossas arrays localmente para reutilização. Para isso simplesmente criamos uma array comum atribuída a variável data.

Em seguida por meio da função np.save( ) podemos de fato exportar os dados de nossa array para um arquivo loca. Note que como parâmetros da função np.save( ) em primeiro lugar repassamos em forma de string um nome para nosso arquivo, seguido do nome da variável que possui a array atribuída para si.

O arquivo gerado será, nesse caso, minha_array.npy.

Carregando uma array do disco local

Da mesma forma que era possível salvar nossa array em um arquivo no disco local, outra possibilidade que temos é de justamente carregar este arquivo para sua utilização.

```
np.load ( 'minha_array.npy' )
```

Através da função np.load( ), bastando passar como parâmetro o nome do arquivo com sua extensão, o carregamento será feito, inclusive já alocando em memória a array para que se possam realizar operações sobre a mesma.

Exemplo de Aplicação da Biblioteca Numpy

- Vetor não ordenado, orientado a objetos

```
import numpy as np

class Vetor :
    def __init__ ( self , tamanho ) :
        self .tamanho = tamanho
        self .ultimo = -1
        self .elementos = np.empty ( self .tamanho , dtype = int )

    def exibe_em_tela ( self ) :
        if self .ultimo == -1 :
            print ( 'Vetor vazio!!!' )
        else :
            for i in range ( self .ultimo + 1 ):
                print ( i , self .elementos [ i ])
```

```
def insere_elemento ( self , elemento ) :
    if self .ultimo == self .tamanho -1 :
        print ( 'Capacidade máxima atingida' )
    else :
        self .ultimo += 1
        self .elementos [ self .ultimo ] = elemento

def pesquisa_elemento ( self , elemento ) :
    for i in range ( self .ultimo + 1 ):
        if elemento == self .elementos [ i ]:
            return i
    return -1

def exclui_elemento ( self , elemento ) :
   posicao = self .pesquisa_elemento ( elemento )
    if posicao == -1 :
        return -1
    else :
        for i in range ( posicao , self .ultimo ):
            self .elementos [ i ] = self .elementos [ i + 1 ]
        self .ultimo -= 1
```

Partindo para prática, vamos simular uma simples aplicação, toda implementada fazendo o uso de recursos da biblioteca Numpy, onde é criada uma estrutura de vetor não ordenado.

Inicialmente, é necessário sempre realizar as devidas importações das bibliotecas, módulos e pacotes que iremos utilizar ao longo de nosso código. Nesse caso, bastando importar a biblioteca numpy, por convenção a referenciando como np.

Em seguida é criada uma classe de nome Vetor, dentro de seu corpo/escopo, é criado um método construtor/inicializador __init__( ) que define o escopo dessa classe para seus objetos, e que por sua

vez receberá obrigatoriamente um dado/valor a ser utilizado como referência para o tamanho fixo deste vetor.

Sendo assim, são criados alguns objetos de classe, onde o primeiro deles é self.tamanho, que recebe o valor atribuído a tamanho. Também é criado um outro objeto de classe de nome self.ultimo, que recebe como atributo o valor inicial -1, que por sua vez define um gatilho a ser acionado quando o vetor atingir sua capacidade máxima de elementos.

Por fim, é criado um último objeto de classe chamado self.elementos, que recebe como atributo inicial uma array do tipo numpy vazia, porém já com tamanho definido baseado no valor do objeto tamanho, assim como já é aproveitada a deixa para estabelecer o tipo de dado que irá obrigatoriamente compor este vetor.

Na sequência é criado um novo método de classe, dessa vez de nome exibe_em_tela( ), que como o próprio nome já sugere, quando instanciado e inicializado, retornará a própria estrutura array desse vetor.

Para isso, inicialmente é criada uma estrutura condicional que verifica se o valor de self.ultimo for igual a -1, então exibe em tela via função print( ) a mensagem 'Vetor vazio!!!'.

Caso essa condição não seja válida, por meio de um laço for é percorrido cada elemento de nosso vetor, a cada ciclo de repetição exibindo o mesmo assim como seu número de índice.

Dando sequência em nosso código, é criado um novo método de classe, dessa vez chamado insere_elemento( ), que recebe como parâmetro um elemento.

Dentro do corpo desta função inicialmente é criada uma estrutura condicional onde se o valor atribuído a self.ultimo for igual ao valor de self.tamanho – 1, é exibido em tela a mensagem 'Capacidade máxima atingida'.

Caso contrário, self.ultimo tem seu valor incrementado em 1 unidade, assim como para self.elementos na posição [self.ultimo] é inserido o novo elemento anteriormente repassado como parâmetro para nossa função insere_elemento( ).

Na sequência é criado um novo método de classe, agora chamado pesquisa_elemento( ) que por sua vez receberá como parâmetro um elemento.

No bloco indentado a esta função temos um laço de repetição for que percorre cada um dos elementos de nosso vetor, validando se o valor atual de elemento é igual a self.elementos na posição [i], caso seja igual, é retornado o último valor atribuído a i, caso não seja igual, é retornado -1.

Por fim, é criado um último método de classe, dessa vez de nome exclui_elemento( ) que receberá por justaposição um elemento como parâmetro.

Dentro do corpo dessa função é criada uma variável local de nome posição, que por sua vez recebe como atributo os dados/valores oriundos da função aninhada self.pesquisa_elemento(elemento).

Em seguida é criada uma estrutura condicional onde se o valor de posição for igual a -1, é retornado -1, caso contrário, é feito o uso de um laço de repetição que percorrerá todos os elementos de posição equiparados com self.ultimo.

A partir deste ponto, self.elementos na posição [i] tem seu valor atualizado com o último valor atribuído a self.elementos em sua posição [i + 1], finalizando com o decremento de sela.ultimo em 1 unidade.

```
base = Vetor ( 10 )
base.exibe_em_tela ()
```

Devolta ao escopo global do código, é criada uma variável de nome base que por sua vez instancia e inicializa a classe Vetor( ),

repassando como argumento para a mesma o valor 10. Em outras palavras, aqui a variável base importa toda a estrutura interna da classe Vetor, assim como define um tamanho fixo de vetor em 10 elementos.

Usando do método base.exibe_em_tela( ) o retorno gerado neste momento é 'Vetor vazio!!!", uma vez que ainda não inserimos elementos no mesmo.

```
base.insere_elemento ( 9 )
base.insere_elemento ( 3 )
base.exibe_em_tela ()
```

Usando do método insere_elemento( ) atrelado a nossa variável base, podemos inserir alguns elementos em nosso vetor. Novamente, usando do método exibe_em_tela( ), nos é retornado o vetor representado por seus elementos.

0 9
1 3

```
print ( base.pesquisa_elemento ( 9 ))
```

Realizando outro tipo de interação com nosso vetor, podemos usar de nosso método pesquisa_elemento( ) parametrizado com o elemento em si para descobrir sua posição de índice no vetor.

```
print ( base.ultimo )
```

Também é possível pesquisar diretamente o último valor atribuído para o objeto ultimo, usando desse retorno, que será um número de índice, para descobrirmos quantos elementos compõe nosso vetor.

```
base.insere_elemento ( 1 )
```

```
base.insere_elemento ( 4 )
base.insere_elemento ( 5 )
base.insere_elemento ( 12 )
base.insere_elemento ( 20 )
base.exibe_em_tela ()
```

Uma vez que nosso vetor esteja definido, assim como para o mesmo não exista nenhum conflito de interação ou até mesmo de sintaxe, podemos manipular este vetor à vontade.

Inserindo alguns elementos via método insere_elemento( ), podemos visualizar nosso vetor instanciando e executando o método exibe_em_tela( ) sempre que necessário.

0 9
1 3
2 1
3 4
4 5
5 12
6 20

```
base.exclui_elemento ( 12 )
base.exibe_em_tela ()
```

Por fim, testando nosso método exclui_elemento( ) será possível ver que quando excluímos um determinado elemento do vetor, os elementos subsequentes serão realocados automaticamente para que as posições de índice se mantenham íntegras.

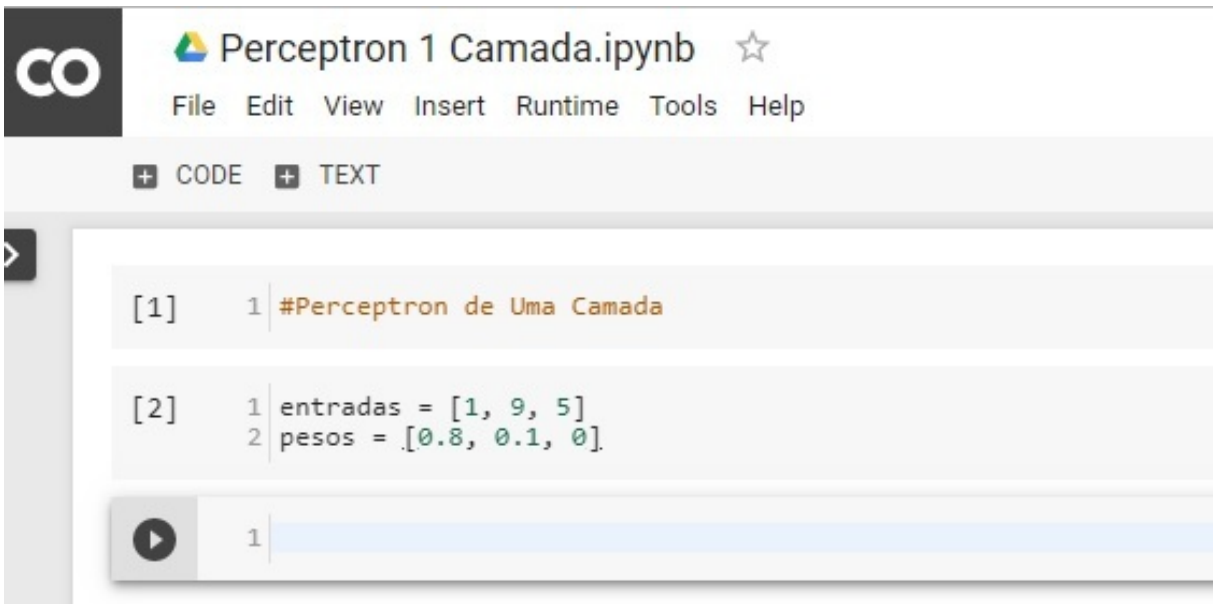
Nesse caso, o retorno será:

0 9
1 3
2 1
3 4
4 5
5 20

- Perceptron de Uma Camada – Modelo Simples

Inicialmente vamos fazer a codificação do algoritmo de um Perceptron, estrutura de dado utilizada em machine learning para abstrair uma rede neural biológica para rede neural artificial.

Sendo assim, pondo em prática o mesmo, através do Google Colaboratory (sinta-se livre para usar o Colaboratory ou o Jupyter da suíte Anaconda), ferramentas que nos permitirão tanto criar este perceptron quanto o executar em simultâneo. Posteriormente em problemas que denotarão maior complexidade estaremos usando outras ferramentas.



Abrindo nosso Colaboratory inicialmente renomeamos nosso notebook para Perceptron 1 Camada.ipynb apenas por convenção. *Ao criar um notebook Python 3 em seu Colaboratory automaticamente será criada uma pasta em seu Drive de nome Colab Notebooks, onde será salva uma cópia deste arquivo para posterior utilização.

Por parte de código apenas foi criado na primeira célula um comentário, na segunda célula foram declaradas duas variáveis de nomes entradas e pesos, respectivamente. Como estamos atribuindo vários valores para cada uma passamos esses valores em forma de lista, pela sintaxe Python, entre chaves e separados por vírgula.

Aqui estamos usando os mesmos valores usados no modelo de perceptron anteriormente descrito, logo, entradas recebe os valores 1, 9 e 5 referentes aos nós de entrada X, Y e Z de nosso modelo assim como pesos recebe os valores 0.8, 0.1 e 0 como no modelo.

```
[3]  1 def soma(e,p):
     2   s = 0
     3   for i in range(3):
     4     s += e[i] * p[i]
     5   return s
```

Em seguida criamos uma função de nome soma que recebe como parâmetro as variáveis temporárias e e p. Dentro dessa função inicialmente criamos uma nova variável de nome s que inicializa com valor 0. Em seguida criamos um laço de repetição que irá percorrer todos valores de e e p, realizar a sua multiplicação e atribuir esse valor a variável s. Por fim apenas deixamos um comando para retornar s, agora com valor atualizado.

```
[4]  1 s = soma(entradas,pesos)
```

```
[5]  1 print(s)
```

```
1.7000000000000002
```

Criamos uma variável de nome s que recebe como atributo a função soma, passando como parâmetro as variáveis entradas e pesos. Executando uma função print( ) passando como parâmetro a variável s finalmente podemos ver que foram feitas as devidas operações sobre as variáveis, retornando o valor de 1.7, confirmando o modelo anterior.

```
[6]  1 def stepFunction(s):
     2   if (s >= 1):
     3     return 1
     4   return 0
```

Assim como criamos a função de soma entre nossas entradas e seus respectivos pesos, o processo final desse perceptron é a criação de nossa função degrau. Função essa que simplesmente irá pegar o valor retornado de soma e estimar com base nesse valor se esse neurônio será ativado ou não.

Para isso simplesmente criamos outra função, agora de nome stepFunction( ) que recebe como parâmetro s. Dentro da função simplesmente criamos uma estrutura condicional onde, se o valor de s for igual ou maior que 1, retornará 1 (referência para ativação), caso contrário, retornará 0 (referência para não ativação).

```
[7]  1 saida = stepFunction(s)
```

```
[8]  1 print(saida)
```

```
1
```

Em seguida criamos uma variável de nome saida que recebe como atributo a função stepFunction( ) que por sua vez tem como parâmetro s. Por fim executamos uma função print( ) agora com saida como parâmetro, retornando finalmente o valor 1, resultando como previsto, na ativação deste perceptron.

Código Completo:

```python
#Perceptron de Uma Camada

entradas = [1, 9, 5]
pesos = [0.8, 0.1, 0]

def soma(e,p):
  s = 0
  for i in range(3):
    s += e[i] * p[i]
  return s

s = soma(entradas,pesos)

def stepFunction(s):
  if (s >= 1):
    return 1
  return 0

saida = stepFunction(s)

print(s)
print(saida)
```

```
1.7000000000000002
1
```

Aprimorando o Código:

Como mencionado nos capítulos iniciais, uma das particularidades por qual escolhemos desenvolver nossos códigos em Python é a questão de termos diversas bibliotecas, módulos e extensões que irão facilitar nossa vida oferecendo ferramentas que não só automatizam, mas melhoram processos no que diz respeito a performance.

Não existe problema algum em mantermos o código acima usando apenas os recursos do interpretador do Python, porém se podemos realizar a mesma tarefa de forma mais reduzida e com maior performance por quê não o fazer.

Sendo assim, usaremos aplicado ao exemplo atual alguns recursos da biblioteca Numpy, de forma a usar seus recursos internos para tornar nosso código mais enxuto e eficiente. Aqui trabalharemos

pressupondo que você já instalou tal biblioteca como foi orientado em um capítulo anterior.

```
[1]    1 import numpy as np
```

Sempre que formos trabalhar com bibliotecas que nativamente não são carregadas e pré-alocadas por nossa IDE precisamos fazer a importação das mesmas. No caso da Numpy, uma vez que essa já está instalada no sistema, basta executarmos o código import numpy para que a mesma seja carregada e possamos usufruir de seus recursos.

Por convenção quando importamos alguns tipos de biblioteca ou módulos podemos as referenciar por alguma abreviação, simplesmente para facilitar sua chamada durante o código. Logo, o comando import numpy as np importa a biblioteca Numpy e sempre que a formos usar basta iniciar o código com np.

```
[2]    1 entradas = np.array([1, 9, 5])
       2 pesos = np.array([0.8, 0.1, 0])
```

Da mesma forma que fizemos anteriormente, criamos duas variáveis que recebem os respectivos valores de entradas e pesos. Porém, note a primeira grande diferença de código, agora não passamos uma simples lista de valores, agora criamos uma array Numpy para que esses dados sejam vetorizados e trabalhados internamente pelo Numpy. Ao usarmos o comando np.array( ) estamos criando um vetor ou matriz, dependendo da situação, para que as ferramentas internas desta biblioteca possam trabalhar com os mesmos.

```
[3]    1 def Soma(e,p):
       2   return e.dot(p)
```

Segunda grande diferença agora é dada em nossa função de Soma, repare que agora ela simplesmente contém como parâmetros e e p, e ela executa uma única linha de função onde ela retornará o

produto escalar de e sobre p. Em outras palavras, o comando e.dot(p) fará o mesmo processo aritmético que fizemos manualmente, de realizar as multiplicações e somas dos valores de e com p, porém de forma muito mais eficiente.

```
[4]  1 s = Soma(entradas, pesos)
     2
     3 def stepFunction(Soma):
     4   if (s >= 1):
     5     return 1
     6   return 0
     7
     8 saida = stepFunction(s)
     9
    10 print(s)
    11 print(saida)
```

```
1.7000000000000002
1
```

Todo o resto do código é reaproveitado e executado da mesma forma, obtendo inclusive como retorno os mesmos resultados (o que é esperado), a diferença de trocar uma função básica, baseada em listas e condicionais, por um produto escalar realizado por módulo de uma biblioteca dedicada a isto torna o processo mais eficiente.

Raciocine que à medida que formos implementando mais linhas de código com mais funções essas pequenas diferenças de performance realmente irão impactar o desempenho de nosso código final.

Código Completo:

```
import numpy as np

entradas = np.array([1, 9, 5])
pesos = np.array([0.8, 0.1, 0])

def Soma(e,p):
  return e.dot(p)

s = Soma(entradas, pesos)

def stepFunction(Soma):
  if (s >= 1):
    return 1
  return 0

saida = stepFunction(s)

print(s)
print(saida)
```

```
1.7000000000000002
1
```

Usando o Spyder 3:

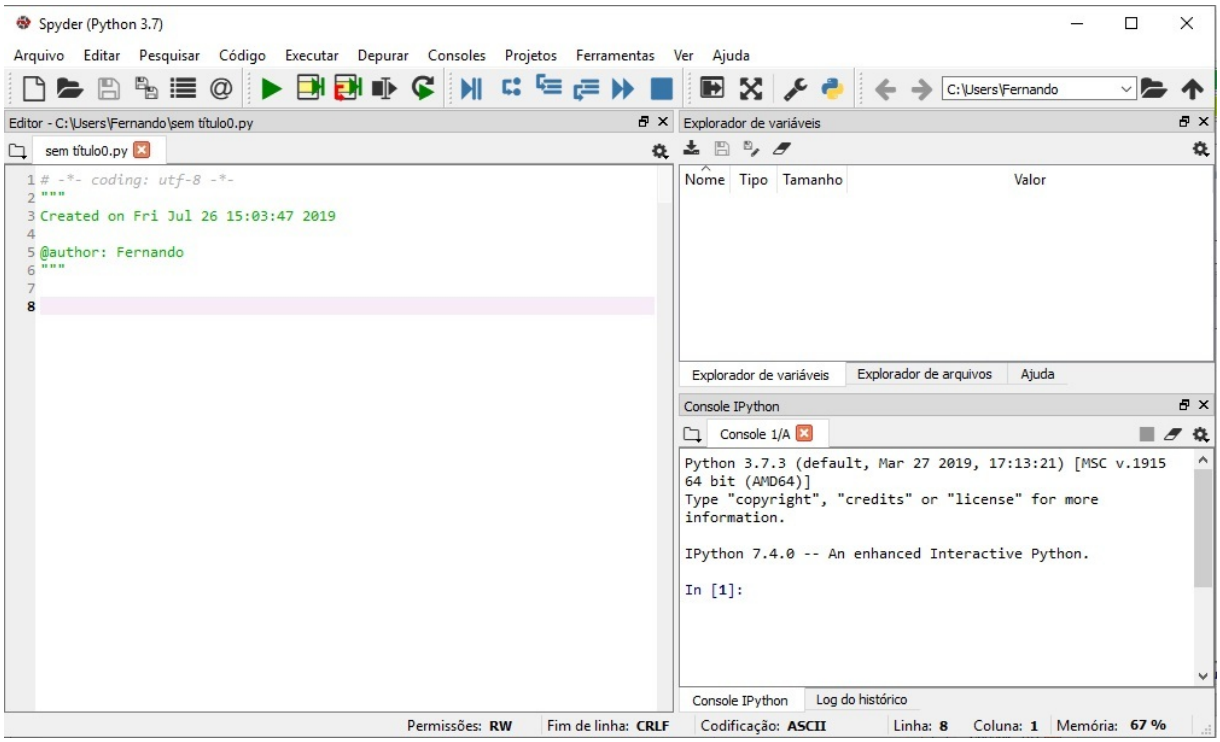
Os códigos apresentados anteriormente foram rodados diretamente no notebook Colab do Google, porém haverão situações de código mais complexo onde não iremos conseguir executar os códigos normalmente a partir de um notebook.

Outra ferramenta bastante usada para manipulação de dados em geral é o Spyder. Dentro dele podemos criar e executar os mesmos códigos de uma forma um pouco diferente graças aos seus recursos.

Por hora, apenas entenda que será normal você ter de se familiarizar com diferentes ferramentas uma vez que algumas situações irão requerer mais ferramentas específicas.

Tratando-se do Spyder, este será o IDE que usaremos para praticamente tudo a partir daqui, graças a versatilidade de por meio dele podermos escrever nossas linhas de código, executá-las

individualmente e em tempo real e visualizar os dados de forma mais intuitiva.



Abrindo o Spyder diretamente pelo seu atalho ou por meio da suíte Anaconda nos deparamos com sua tela inicial, basicamente o Spyder já vem pré-configurado de forma que possamos simplesmente nos dedicar ao código, talvez a única configuração que você deva fazer é para sua própria comodidade modificar o caminho onde serão salvos os arquivos.

A tela inicial possui a esquerda um espaço dedicado ao código, e a direita um visualizador de variáveis assim como um console que mostra em tempo real certas execuções de blocos de código.

Apenas como exemplo, rodando este mesmo código criado anteriormente, no Spyder podemos em tempo real fazer a análise das operações sobre as variáveis, assim como os resultados das mesmas via terminal.

Perceptron 1 Camada Otimizado.py*

```
1 import numpy as np
2
3 entradas = np.array([1, 9, 5])
4 pesos = np.array([0.8, 0.1, 0])
5
6 def Soma(e,p):
7   return e.dot(p)
8
9 s = Soma(entradas, pesos)
10
11 def stepFunction(Soma):
12   if (s >= 1):
13     return 1
14   return 0
15
16 saida = stepFunction(s)
17
18 print(s)
19 print(saida)
20
```

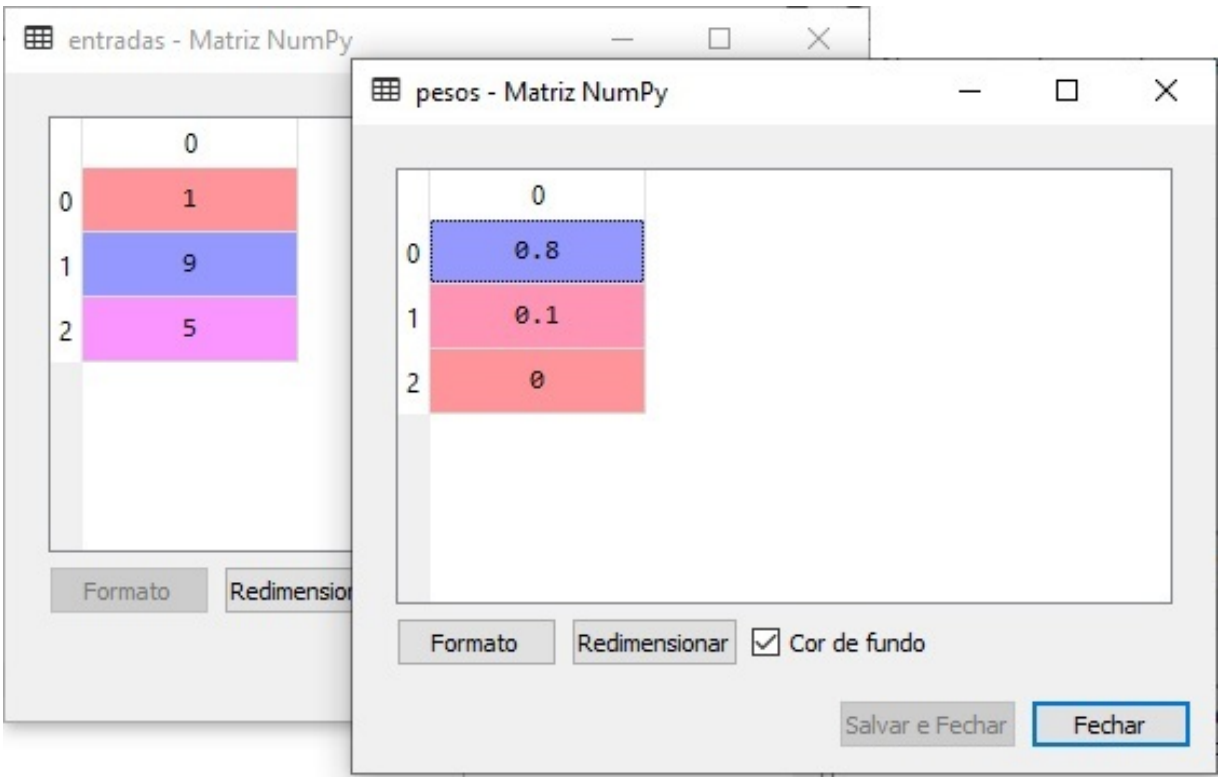
Explorador de Variáveis:

Explorador de variáveis

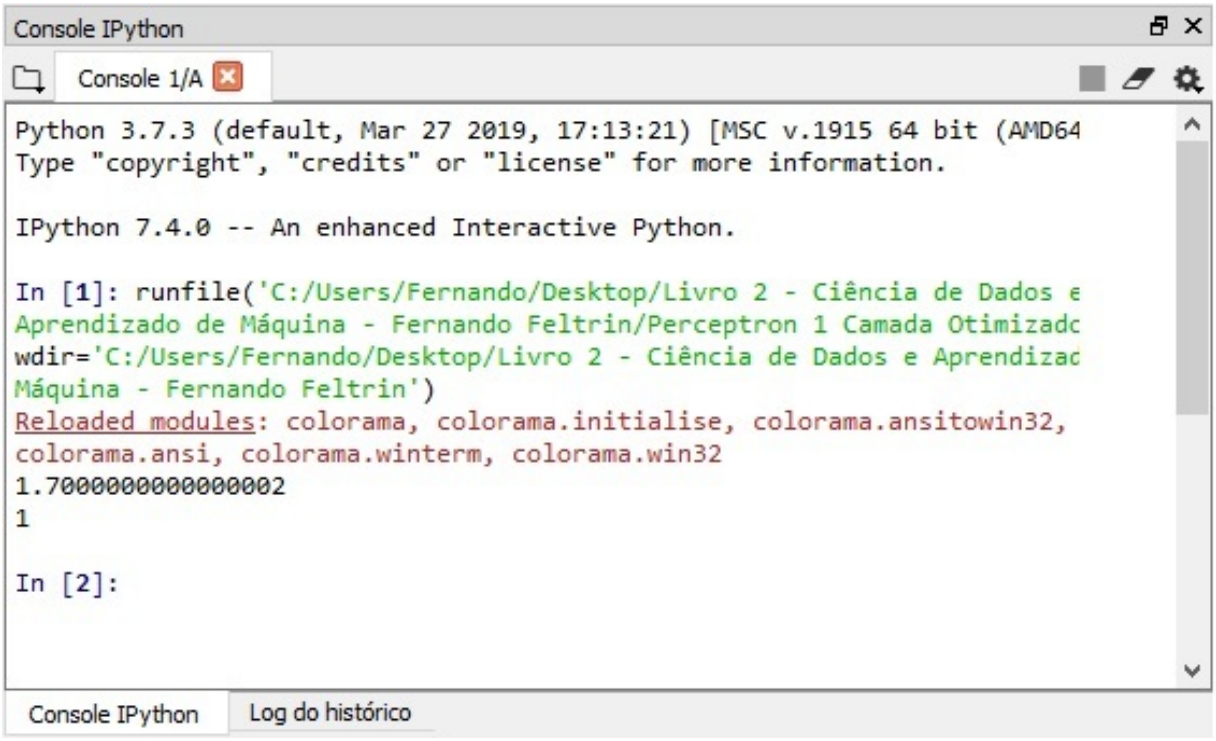
Nome	Tipo	Tamanho	Valor
entradas	int32	(3,)	[1 9 5]
pesos	float64	(3,)	[0.8 0.1 0.]
s	float64	1	1.7000000000000002
saida	int	1	1

Explorador de variáveis | Explorador de arquivos | Ajuda

Visualizando Variáveis:


entradas - Matriz NumPy
0
0
1
1
9
2
5
Formato
Redimensior
pesos - Matriz NumPy
0
0
0.8
1
0.1
2
0
Formato
Redimensionar
Cor de fundo
Salvar e Fechar
Fechar

Terminal:


Console IPython
Console 1/A
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64
Type "copyright", "credits" or "license" for more information.
IPython 7.4.0 -- An enhanced Interactive Python.
In [1]: runfile('C:/Users/Fernando/Desktop/Livro 2 - Ciência de Dados e
Aprendizado de Máquina - Fernando Feltrin/Perceptron 1 Camada Otimizadc
wdir='C:/Users/Fernando/Desktop/Livro 2 - Ciência de Dados e Aprendizac
Máquina - Fernando Feltrin')
Reloaded modules: colorama, colorama.initialise, colorama.ansitowin32,
colorama.ansi, colorama.winterm, colorama.win32
1.7000000000000002
1
In [2]:
Console IPython
Log do histórico

Conclusão

Apenas fazendo um encerramento, vimos neste pequeno livro algumas das possibilidades de operações com arrays por meio da biblioteca Numpy. A linguagem Python é muito versátil no que diz respeito tanto da sua flexibilidade quando usada apenas com recursos nativos, quanto quando usada com bibliotecas desenvolvidas pela comunidade com seus respectivos recursos adicionais.

Dependendo muito do nicho que você faz uso da linguagem Python, certamente existirão bibliotecas específicas para tal propósito, aumentando ainda mais o número de recursos oferecidos e passíveis de implementação.

Espero que você tenha entendido por meio dos exemplos apresentados neste livro, as principais funcionalidades que a biblioteca Numpy nos oferece. Como dito nos capítulos iniciais, existe uma enorme gama de funções predefinidas nessa biblioteca, aqui pelas limitações de um livro focamos apenas nas principais, nas mais comumente utilizadas.

Muito obrigado pela aquisição deste material, espero que de fato ele tenha contribuído de alguma forma para seu aprendizado. Caso sim, considere adquirir outro de meus livros, todos possuem a mesma didática e abordam temas desde o básico de Python até o que existe de mais avançado dentro de Redes Neurais Artificiais aplicadas em Inteligência Artificial.

Novamente, muito obrigado, lhe desejo sucesso em suas idealizações.

Códigos Completos

Instalação das dependências

```
pip install numpy
```

Importando a biblioteca numpy

```
import numpy as np
```

Criando uma array numpy

```
data = np.array ([ 2 , 4 , 6 , 8 , 10 ])

print ( data )
print ( type ( data ))
print ( data.shape )
```

Criando automaticamente uma array numpy com dados ordenados.

```
data = np.arange ( 15 )

print ( data )
```

Criando automaticamente uma array numpy com dados aleatórios (float entre 0 e 1)

```
data = np.random.rand ( 15 )

print ( data )
```

Criando uma array numpy com dados aleatórios (int entre 0 e o valor especificado).

```
data = np.random.randint ( 10 , size = 10 )
# números de 0 a 10, size define quantos elementos serão gerados
```

```
print ( data )
```

Criando uma array numpy com dados aleatórios (float entre 0 e 1)

```
data = np.random.random (( 2 ,  2 ))

print ( data )
```

Criando uma array numpy composta de zeros

```
data = np.zeros (( 3 ,  4 ))
# 3 linhas e 4 colunas

print ( data )
```

Criando uma array numpy composta de números um

```
data = np.ones (( 3 ,  4 ))
# 3 linhas e 4 colunas

print ( data )
```

Criando uma array numpy com valores vazios.

```
data = np.empty (( 3 ,  4 ))
# haverá um valor, representado em notação científica, algo muito próximo ao zero absoluto.

print ( data )
```

Criando uma array numpy de números negativos.

```
data = np.arange ( 10 ) * -1

print ( data )
```

Criando uma array numpy unidimensional

```
data = np.random.randint ( 5 ,  size =  10 )
# números de 0 a 5, size define quantos elementos serão gerados

print ( data )
print ( data.shape )
```

Criando uma array numpy bidimensional

```
data2 = np.random.randint ( 5 ,  size =  ( 3 ,  4 ))
# números de 0 a 5, distribuidos em 3 linhas e 4 colunas

print ( data2 )
print ( data2.shape )
```

Criando uma array numpy tridimensional

```
data3 = np.random.randint ( 5 ,  size =  ( 5 ,  3 ,  4 ))
# números de 0 a 5, 5 camadas cada uma com 3 linhas e 4 colunas

print ( data3 )
print ( data3.shape )
```

Criando uma array numpy a partir de elementos de uma lista

```
data4 = np.array ([[ 1 ,  2 ,  3 ,  4 ],  [ 1 ,  3 ,  5 ,  7 ]])

print ( data4 )
```

Verificando o tipo de dado de uma array

```
print ( type ( data3 ))
print ( data3.dtype )
```

Verificando o tamanho de uma array

```
data3 = np.random.randint ( 5 ,  size =  ( 5 ,  3 ,  4 ))

print ( data3.ndim )
# número de dimensões

print ( data3.shape )
# formato de cada dimensão

print ( data3.size )
# tamanho total da matriz
```

Verificando o tamanho em bytes de cada item e de toda uma array

```
print ( f 'Cada elemento possui  { data3.itemsize }  bytes' )

print ( f 'Toda a matriz possui  { data3.nbytes }  bytes' )
```

Consultando o elemento de maior valor em uma array

```
data = np.arange ( 15 )

print ( data )
print ( data. max ())
```

Consultando um elemento através de seu índice (unidimensional)

```
data = np.arange ( 15 )

print ( data )
print ( data [ 8 ])
```

```
data

print ( data )
```

```
print ( data [ -5 ])
# 5º elemento a contar do fim para o começo
```

Consultando um elemento através de seu índice (multidimensional)

```
data2 = np.random.randint ( 5 ,  size =  ( 3 ,   4 ))

print ( data2 )
print ( data2 [ 0 ,   2 ])
# linha 0, coluna 2 (lembrando que o índice para coluna também começa em 0, 0-1-2-3)
```

Consultando um intervalo de elementos em uma array

```
data

print ( data )

print ( data [: 5 ])
# apenas os 5 primeiros

print ( data [ 3 :])
# do terceiro elemento em diante (elemento, e não índice)

print ( data [ 4 : 8 ])
# do quarto elemento até o oitavo (4º, 5º, 6º 7º e 8º elemento)

print ( data [:: 2 ])
# de dois em dois elementos

print ( data [ 3 ::])
# pula os 3 primeiros elementos e imprime o resto
```

Modificando dados/valores manualmente por meio de seu índice

```
data2
```

```
print ( data2 )

data2 [ 0 ,  0 ] = 10
# elemento situado na posição 0x0 do índice passará a ter o valor atribuído 10

print ( data2 )
```

Inserindo elementos diferentes de int

```
data2

data2 [ 1 ,  1 ] = 6.82945
# mesmo declarado como float será convertido em int

print ( data2 )
```

Definindo manualmente o tipo de dados de uma array

```
data2 = np.array ([ 8 ,  -3 ,  5 ,  9 ], dtype = 'float' )
# todos elementos serão convertidos para float

print ( data2 )
print ( type ( data2 [ 0 ]))
```

Criando uma array numpy de intervalos igualmente distribuídos

```
data = np.linspace ( 0 ,  1 ,  7 )
# 7 números, gerados de 0 a 1, com intervalo igual entre cada número gerado

print ( data )
```

Criando uma array numpy com formato definido manualmente

```
data5 = np.arange ( 8 )
```

```
print ( data5 )

data5 = data5.reshape ( 2 ,  4 )
# a multiplicação dessas dimensões deve ser o número de elementos. ex 2 x 4 = 8

print ( data5 )
```

Criando uma array numpy com tamanho predefinido em variável

```
tamanho =  10

data6 = np.random.permutation ( tamanho )
# matriz de 10 elementos conforme variável tamanho define

print ( data6 )
```

Operadores lógicos em arrays numpy (todos elementos)

```
data7 = np.arange ( 10 )

print ( data7 )
print ( data7 >  3 )
# irá verificar cada elemento da matriz, retornando true para os que forem maior que 3 e false para os que forem menor.
```

Operadores lógicos em arrays numpy (elemento específico)

```
data7 = np.arange ( 10 )

print ( data7 )
print ( data7 [ 4 ]  >  5 )
# o elemento e é maior que 5?
```

Operadores aritméticos em arrays numpy

```
data8 =  [ 1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ]
print ( data8 )
```

```
data8 = np.array ( data8 )
# após convertido para array o operador + sempre fará a soma

print ( data8 +  10 )
```

Criando uma matriz diagonal

```
data9 = np.diag (( 1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,  8 ))
# os elementos serão distribuídos em uma coluna diagonal

print ( data9 )
```

Criando padrões duplicados

```
data10 = np.tile ( np.array ([[ 9 ,  4 ],  [ 3 ,  7 ]]),  4 )
# 9, 4, elementos da primeira linha, 3, 7 da segunda, duplicados 4 vezes na mesm
a linha

print ( data10 )
```

```
data10 = np.tile ( np.array ([[ 9 ,  4 ],  [ 3 ,  7 ]]),  ( 2 , 2 ))
# mesmo exemplo que o anterior, mas duplicados linha e coluna

print ( data10 )
```

Somando um valor a cada elemento de uma array

```
data11 = np.arange ( 0 ,  15 )

print ( data11 )

data11 = np.arange ( 0 ,  15 )  +  1
```

```
# irá somar 1 a cada elemento

print ( data11 )
```

```
data11 = np.arange ( 0 ,  30 , 3 )  +  3
# somando 3 ao valor de cada elemento, de 3 em 3 elementos

print ( data11 )
```

Realizando a soma de arrays

```
d1 = np.array ([  3 ,  6 ,  9 ,  12 ,  15 ])
d2 = np.array ([  1 ,  2 ,  3 ,  4 ,  5 ])

d3 = d1 + d2

print ( d3 )
```

Realizando a subtração de arrays

```
d1 = np.array ([  3 ,  6 ,  9 ,  12 ,  15 ])
d2 = np.array ([  1 ,  2 ,  3 ,  4 ,  5 ])

d3 = d1 - d2

print ( d3 )
```

Realizando a divisão de arrays

```
d1 = np.array ([  3 ,  6 ,  9 ,  12 ,  15 ])
d2 = np.array ([  1 ,  2 ,  3 ,  4 ,  5 ])

d3 = d1 / d2

print ( d3 )
```

Realizando a multiplicação de arrays

```
d1 = np.array ([  3 ,   6 ,   9 ,  12 ,  15 ,  18 ,  21 ,  24 ,  27 ,  30 ])
d2 = np.array ([  1 ,   2 ,   3 ,   4 ,   5 ,   6 ,   7 ,   8 ,   9 ,  10 ])

d3 = d1 * d2

print ( d3 )
```

Operações lógicas entre arrays

```
d1 = np.array ([  3 ,   6 ,   9 ,  12 ,  15 ,  18 ,  21 ])
d2 = np.array ([  1 ,   2 ,   3 ,   4 ,   5 ,  20 ,  25 ])

d3 = d2 > d1
# os elementos de d2 são maiores do que os de d1?

print ( d3 )
```

Transposição de arrays

```
arr1 = np.array ([[ 1 ,  2 ,  3 ],  [ 4 ,  5 ,  6 ]])

print ( arr1 )
print ( arr1.shape )

arr1_transposta = arr1.transpose ()
# realizará a conversão de linhas para colunas e vice-versa

print ( arr1_transposta )
print ( arr1_transposta.shape )
```

Salvando uma array no disco local

```
data = np.array ([  3 ,   6 ,   9 ,  12 ,  15 ])
```

```
np.save ( 'minha_array' ,  data )
# irá gerar um arquivo com extensão .npy
```

Carregando uma array do disco local

```
np.load ( 'minha_array.npy' )
```

[Python do ZERO à Programação Orientada a Objetos]

[Programação Orientada a Objetos com Python]

[Ciência de Dados e Aprendizado de Máquina]

[Inteligência Artificial com Python]

[Redes Neurais Artificiais com Python]

[Análise Financeira com Python]

[Arrays com Python + Numpy]

[Tópicos Avançados em Python]

[Visão Computacional em Python]

Qualquer errata, dúvida, crítica ou sugestão, fique à vontade para entrar em contato diretamente através do e-mail fernando2rad@gmail.com.