NumPy-Intro

January 1, 2025

1 Intro To NumPy

- numpy is Python library for fast array computing (as fast as C and Fortran) and used in every field of science and engineering
- offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more
- foundation of scientific Python and PyData ecosystems such as:
  - Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science packages
- the heart of NumPy is **ndarray**, a homogenous n-dimensional array object, with methods to efficiently operate on it
- Beginners Guide
- NumPy Fundamentals

1.1 Installation

- can use conda or pip

```
conda config --env --add channels conda-forge
conda install numpy
```

```
pip install numpy
```

1.2 import NumPy

- must import numpy library to use in Python script; typical usage is:

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

[3]:
```python
print(np.__version__)
```

```
1.19.1
```

[2]:
```python
array = np.arange(6)
```

[3]:
```python
array.shape
```

[3]:
```
(6,)
```

[4]:
```python
array
```

[4]: array([0, 1, 2, 3, 4, 5])

1.3 Difference between a Python list and a NumPy array

- NumPy array data has same type (homogenous)
- provides enourmous speed on mathematical operation that are meant to be performed on arrays
- Python list can contain different data types within a single list (heterogenous)
  - much slower and inefficienet in operations

1.4 NumPy array

- central data structure of the NumPy library
- grid of elments that can be indexed in various ways
- the elements are of the same type, referred to as the array **dtype**
- the **rank** of the array is the number of dimensions
- the **shape** of the array is a tuple of integers giving the size of the array along each dimension
- can initialize NumPy arrays from Python lists

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

[6]:
```
b = np.array([[1, 2, 3, 4], [10, 20, 30, 40], [100, 200, 300, 400]])
```

[8]:
```
b.shape
```

[8]: (3, 4)

[7]:
```
# accessing np array is similar to Python list using 0-based indices
print(a[0])
```

```
1
```

[10]:
```
print(b)
```

```
[[  1   2   3   4]
 [ 10  20  30  40]
 [100 200 300 400]]
```

[9]:
```
print(b[2][0])
```

```
100
```

1.4.1 Types of array

- **1-D** array is also called **vector**
  - no difference between row and column vectors
- **2-D** array is also called **matrix**
- **3-d** and higher dimensional arrays are also called **tensor**

1.4.2 Attributes of an array

- array is usually a fixed-size container of items of the same type and size
- the number of dimensions and items in an array is defined byt its shape
- the shape is a tuple that specify the sizes of each dimension
- NumPy dimensions are called **axes**
- the *b* NumPy **ndarray** is a 2-d matrix
- the *b* array has 2 axes
- the first axis (row) has length of 3 and the second axis (column) has a length of 4

[11]:
```
b
```

[11]:
```
array([[  1,   2,   3,   4],
       [ 10,  20,  30,  40],
       [100, 200, 300, 400]])
```

1.5 Creating basic array

- various ways; primary is by using **np.array()**

[16]:
```
a = np.array([1, 2, 3])
```

[17]:
```
a
```

[17]:
```
array([1, 2, 3])
```

[18]:
```
# create and initialize elements with 0s
a = np.zeros(4)
```

[19]:
```
a
```

[19]:
```
array([0., 0., 0., 0.])
```

[20]:
```
# create an initialize elements with 1s
a = np.ones(5)
```

[21]:
```
a
```

[21]:
```
array([1., 1., 1., 1., 1.])
```

[22]:
```
# create an empty array with random values; make sure to fill the array with␣
↪actual elements
a = np.empty(2)
```

[23]:
```
a
```

[23]:
```
array([2.05833592e-312, 2.33419537e-312])
```

[24]:
```
# use arange(start, stop, step)
np.arange(2, 9, 2)
```

[24]:
```
array([2, 4, 6, 8])
```

[25]:
```
# create an array with values that are spaced linearly in a specified interval
np.linspace(0, 10, num=5)
```

[25]:
```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

[27]:
```
# specify datatype; default is np.float64
np.ones(5, dtype=np.int64)
```

[27]:
```
array([1, 1, 1, 1, 1])
```

1.6 Adding, removing, and sorting elements

- https://numpy.org/devdocs/reference/generated/numpy.sort.html#numpy.sort
- `np.sort(a, axis=-1, kind=None, order=None)` - array a to be sorted and return the sorted ndarray
  - axis : default-1 sorts along the last axis
  - kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, default is quicksort
  - order: str or list of str where str is field name or list of field names

[29]:
```
a = np.array([3, 1, 2, 4])
```

[30]:
```
a.sort()
```

[31]:
```
a
```

[31]:
```
array([1, 2, 3, 4])
```

[33]:
```
b = np.array([5, 6, 7, 8])
```

[34]:
```
np.concatenate((a, b))
```

[34]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

[37]:
```
np.concatenate((a, b), axis=0)
```

[37]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

[38]:
```
c = np.array([7, 8, 9, 10])
```

[39]:
```
np.concatenate((a, b, c))
```

[39]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  7,  8,  9, 10])
```

[45]:
```
# concatenate 2-d array
matrix = np.concatenate(([a], [b], [c]))
```

[46]:
```
matrix
```

[46]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 7,  8,  9, 10]])
```

1.7 know the shape and size of array

- ndarray.shape, ndarray.size, ndarray.ndim

[47]:
```
matrix.shape
```

[47]:
```
(3, 4)
```

[48]:
```
matrix.size
# product of the elements of array's shape
```

[48]:
```
12
```

[49]:
```
matrix.ndim
# number of axes or dimensions
```

[49]:
```
2
```

1.8 Indexing and slicing

- NumPy arrays can be sliced the same way as Python lists

[50]:
```
data = np.array([1, 2, 3])
```

[51]:
```
data[1]
```

[51]:
```
2
```

[53]:
```
data[1:]
```

[53]:
```
array([2, 3])
```

[54]:
```
data[-1]
```

[54]:
```
3
```

[55]:
```
# slice array with certain conditions
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

[66]:
```
a
```

[66]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

[57]:
```
# print values in the array that are less than 5 as a 1-d array
print(a[a < 5])
```

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

[64]:
```
# select numbers that are equal to or greater than 5; use that condition to␣
↪index an array
# keeps the original dimension of the array
five_up = a >=5
```

[65]:
```
five_up
```

[65]:
```
array([[False, False, False, False],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

[67]:
```
# select elements that satisfiy two conditions using & and | operators
c = a[(a>2) &(a<11)]
```

[68]:
```
c
```

[68]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

1.9 basic operations on arrays

- `+` - add two arrays' corresponding elements
- `-` - subtract one array from another's corresponding elements
- `*` - multiply one array by another's corresponding elements
- `/` - divide one array by another's corresponding elements

[69]:
```
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
```

[71]:
```
data
```

[71]:
```
array([1, 2])
```

[72]:
```
ones
```

[72]:
```
array([1, 1])
```

[70]:
```
data + ones
```

[70]:
```
array([2, 3])
```

[73]:
```
data - ones
```

[73]: array([0, 1])

[74]:
```
data / ones
```

[74]: array([1., 2.])

[75]:
```
data.sum()
```

[75]: 3

[101]:
```
# you specifiy the axis on 2-d array
b = np.array([[1, 1], [0.5, 0.5]])
```

[102]:
```
# sum the rows
b.sum(axis=0)
```

[102]: array([1.5, 1.5])

[103]:
```
# sum the columns
b.sum(axis=1)
```

[103]: array([2., 1.])

[104]:
```
b.min()
```

[104]: 0.5

[105]:
```
b.max()
```

[105]: 1.0

[106]:
```
b.sum()
```

[106]: 3.0

[107]:
```
# find min on each column
b.min(axis=0)
```

[107]: array([0.5, 0.5])

[108]:
```
# find min on each row
b.min(axis=1)
```

[108]: array([1. , 0.5])

1.10 Broadcasting

- an operation between a vector and a scalar applies to all the elements in vector

[85]:
```
data = np.array([1.0, 2.0, 3.0])
```

[86]:
```
data * 1.6
```

[86]:
```
array([1.6, 3.2, 4.8])
```

[87]:
```
data + 1.1
```

[87]:
```
array([2.1, 3.1, 4.1])
```

[88]:
```
data / 2
```

[88]:
```
array([0.5, 1. , 1.5])
```

[89]:
```
data - 1
```

[89]:
```
array([0., 1., 2.])
```

1.11 Matrix computation

- linear-algebra based computation and more…
- https://numpy.org/doc/stable/reference/routines.linalg.html

[3]:
```
A = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

[4]:
```
A
```

[4]:
```
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

[5]:
```
B = np.array([[2, 2, 2], [2, 2, 2], [2, 2, 2]])
```

[6]:
```
B
```

[6]:
```
array([[2, 2, 2],
       [2, 2, 2],
       [2, 2, 2]])
```

[7]:
```
A + B
```

[7]:
```
array([[3, 4, 5],
       [3, 4, 5],
       [3, 4, 5]])
```

[8]:
```
A - B
```

[8]:
```
array([[-1,  0,  1],
       [-1,  0,  1],
       [-1,  0,  1]])
```

[9]:
```
A * B
```

[9]:
```
array([[2, 4, 6],
       [2, 4, 6],
       [2, 4, 6]])
```

[10]:
```
A / B
```

[10]:
```
array([[0.5, 1. , 1.5],
       [0.5, 1. , 1.5],
       [0.5, 1. , 1.5]])
```

[11]:
```
C = np.dot(A, B)
```

[12]:
```
C
```

[12]:
```
array([[12, 12, 12],
       [12, 12, 12],
       [12, 12, 12]])
```

1.12 Transposing and reshaping a matrix

[111]:
```
data = np.arange(1, 7, 1)
```

[112]:
```
data
```

[112]:
```
array([1, 2, 3, 4, 5, 6])
```

[119]:
```
# 2x3 matrix
X = data.reshape(2, 3)
```

[120]:
```
X
```

[120]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[117]:
```
# 3x2 matrix
data.reshape(3, 2)
```

[117]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[121]:
```
X.transpose()
```

[121]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

[122]:
```
# flatten n-d array to 1-d array
X.flatten()
```

[122]:
```
array([1, 2, 3, 4, 5, 6])
```

1.13 mathematical formulas

- MeanSquareError $= \frac{1}{n}\sum_{i=1}^{n}(Y\_prediction_i - Y_i)^2$

[123]:
```
predictions = np.ones(3)
labels = np.arange(1, 4)
```

[128]:
```
print(predictions, labels)
```

```
[1. 1. 1.] [1 2 3]
```

[126]:
```
error = 1/len(predictions)*np.sum(np.square(predictions-labels))
```

[127]:
```
print(f'supervised ML error= {error}')
```

```
supervised ML error= 1.6666666666666665
```

[ ]:
```

```