NumPy Crash Course

https://www.youtube.com/watch?v=9JUAPgtkKpI

https://github.com/python-engineer/python-engineer-notebooks/blob/master/general/Numpy%20Crash%20Course.ipynb

Learn NumPy in this complete Crash Course! I show you all the essential functions of NumPy and some tricks and useful methods.
NumPy is the core library for scientific computing in Python.
It is essential for any **data science** or **machine learning** algorithms.

Outline

1. NumPy Introduction
2. Installation and Basics
3. Array vs List
4. Dot Product
5. Speed Test array vs list
6. Multidimensional (nd) arrays
7. Indexing/Slicing/Boolean Indexing
8. Reshaping
9. Concatenation
10. Broadcasting
11. Functions and Axis
12. Datatypes
13. Copying
14. Generating arrays
15. Random numbers
16. Linear Algebra (Eigenvalues / Solving Linear Systems)
17. Loading CSV files

Installation

Installation with *pip* or *Anaconda*.

```
$ pip install numpy
or
$ conda install numpy
```

```
import numpy as np
```

```
np.__version__
```

```
'1.19.5'
```

Numpy is the core library in all scientifig packages.
the central object in the numpy library is the numpy array.
Numpy = library for linear algebra and probability

scalar vector matrix

use cases:

- dot product/inner product
- matrix multiplication (shape must match)
- element wise matrix product
- linear systems
- inverse
- determinant
- choose random numbers (e.g. gaussian/uniform)
- working with images represented as array

Applications:

- Machine Learning
- Deep Learning
- scikit-learn
- Matplotlib
- Pandas

```
# array
a = np.array([1,2,3,4,5])
print(a)
print(a.shape) # shape of the array
print(a.dtype) # type of the elements

print(a.ndim)
print(a.size) # total number of elements
print(a.itemsize) # the size in bytes of each element
```

```
[1 2 3 4 5]
(5,)
int64
1
5
8
```

```
# essential methods
a = np.array([1,2,3])
# access and change elements
print(a[0])
a[0] = 5
print(a)

# elementwise math operations
b = a * np.array([2,0,2])
print(b)

print(a.sum())
```

```
print(a.sum())
```

```
1
[5 2 3]
[10  0  6]
10
```

```
# array vs list
l = [1,2,3]
a = np.array([1,2,3])
print(l)
print(a)

# adding new item
l.append(4)
#a.append(4) error: size of array is fixed

# there are ways to add items, but this essentially creates new arrays
l2 = l + [5]
print(l2)

a2 = a + np.array([4])
print(a2) # this is called broadcasting, adds 4 to each element

# vector addidion (this is technically correct compared to broadcasting)
a3 = a + np.array([4,5,6])
print(a3)

#a3 = a + np.array([4,5]) # error, can't add vectors of different sizes

# multiplication
l2 = 2 * l # list l repeated 2 times, same a l+l
print(l2)
a3 = 2 * a # multiplication for each element
print(a3)

# modify each item in the list
l2 = []
```

```
for i in l:
    l2.append(i**2)
print(l2)
# or list comp
l2 = [i**2 for i in l]
print(l2)

a2 = a**2 # -> squares each element!
print(a2)

# Note: function applied to array usually operates element wise
a2 = np.sqrt(a) # np.exp(a), np.tanh(a)
print(a2)
a2 = np.log(a)
print(a2)

# Note: np array exists specifically to do math
```

```
[1, 2, 3]
[1 2 3]
[1, 2, 3, 4, 5]
[5 6 7]
[5 7 9]
[1, 2, 3, 4, 1, 2, 3, 4]
[2 4 6]
[1, 4, 9, 16]
[1, 4, 9, 16]
[1 4 9]
[1.         1.41421356 1.73205081]
[0.         0.69314718 1.09861229]
```

```
# dot product/inner product
a = np.array([1,2])
b = np.array([3,4])

# sum of the products of the corresponding entries
# multiply each corresponding elements and then take the sum

#
```

```
# cumbersome way for lists
dot = 0
for i in range(len(a)):
    dot += a[i] * b[i]
print(dot)

# easy with numpy :)
dot = np.dot(a,b)
print(dot)

# step by step manually
c = a * b
print(c)
d = np.sum(c)
print(d)

# most of these functions are also instance methods
dot = a.dot(b)
print(dot)
dot = (a*b).sum()
print(dot)

# in newer versions
dot = a @ b
print(dot)
```

```
11
11
[3 8]
11
11
11
11
```

```
# speed test lists vs array
from timeit import default_timer as timer

a = np.random.randn(1000)
```

```
b = np.random.randn(1000)

A = list(a)
B = list(b)

T = 1000

def dot1():
    dot = 0
    for i in range(len(A)):
        dot += A[i]*B[i]
    return dot

def dot2():
    return np.dot(a,b)

start = timer()
for t in range(T):
    dot1()
end = timer()
t1 = end-start

start = timer()
for t in range(T):
    dot2()
end = timer()
t2 = end-start

print('Time with lists:', t1)
print('Time with array:', t2)
print('Ratio', t1/t2)
```

```
Time with lists: 0.3177917980000302
Time with array: 0.001636972000028436
Ratio 194.13392409553114
```

```
# nd arrays = multidimensional arrays
# (matrix class exists but not recommended to use)
```

```
# (matrix class exists but not recommended to use)
a = np.array([[1,2], [3,4]])
print(a)
print(a.shape)

# row first, then columns
print(a[0])
print(a[0][0])
print(a[0,0])

# slicing
print(a[:,0]) # all rows in col 0
print(a[0,:]) # all columns in row 0

# transpose
print(a.T)

# matrix multiplication
b = np.array([[3, 4], [5,6]])
c = a.dot(b)
print(c)
d = a * b # elementwise multiplication
print(d)

# inner dimensions must match!
b = np.array([[1,2,3], [4,5,6]])
#c = a.dot(b.T)
#print(c)

# determinant
c = np.linalg.det(a)
print(c)

# inverse
c = np.linalg.inv(a)
print(c)

# diag
```

```
c = np.diag(a)
print(c)

# diag on a vector returns diagonal matrix (overloaded function)
c = np.diag([1,4])
print(c)
```

```
[[1 2]
 [3 4]]
(2, 2)
[1 2]
1
1
[1 3]
[1 2]
[[1 3]
 [2 4]]
[[13 16]
 [29 36]]
[[ 3  8]
 [15 24]]
-2.0000000000000004
[[-2.   1. ]
 [ 1.5 -0.5]]
[1 4]
[[1 0]
 [0 4]]
```

```
# Array indexing
# Slicing: Similar to Python lists, numpy arrays can be sliced.
# Since arrays may be multidimensional, you must specify a slice for each
# dimension of the array:
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)

# Integer array indexing
b = a[0,1]
print(b)

# Slicing
```

```
# Slicing
row0 = a[0,:]
print(row0)

col0 = a[:, 0]
print(col0)

slice_a = a[0:2,1:3]
print(slice_a)

# indexing starting from the end: -1, -2 etc...
last = a[-1,-1]
print(last)
```

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

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

# same shape with True or False for the condition
bool_idx = a > 2
print(bool_idx)

# note: this will be a rank 1 array!
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
# np.where(): same size with modified values
b = np.where(a>2, a, -1)
print(b)

# fancy indexing: access multiple indices at once
a = np.array([10,19,30,41,50,61])
print(a)
b = a[[1,3,5]]
print(b)
even = np.argwhere(a%2==0).flatten()
print(even)
a_even = a[even]
print(a_even)
```

```
[[1 2]
 [3 4]
 [5 6]]
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
[3 4 5 6]
[[-1 -1]
 [ 3  4]
 [ 5  6]]
[10 19 30 41 50 61]
[19 41 61]
[0 2 4]
[10 30 50]
```

```
# reshape
a = np.arange(1, 7)
print(a)

b = a.reshape((2, 3)) # error if shape cannot be used
print(b)

c = a.reshape((3, 2)) # 3 rows, 2 columns
print(c)
```

```
# newaxis is used to create a new axis in the data
# needed when model require the data to be shaped in a certain manner
print(a.shape)
d = a[np.newaxis, :]
print(d)
print(d.shape)

e = a[:, np.newaxis]
print(e)
print(e.shape)
```

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

```
# concatenation
a = np.array([[1, 2], [3, 4]])
print(a)
b = np.array([[5, 6]])
print(b)
c = np.concatenate((a, b), axis=None)
print(c)
d = np.concatenate((a, b), axis=0)
print(d)
e = np.concatenate((a, b.T), axis=1)
```

```
e = np.concatenate((a, b.T), axis=1)
print(e)

# hstack: Stack arrays in sequence horizontally (column wise). needs a tuple
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
c = np.hstack((a,b))
print(c)
a = np.array([[1,2], [3,4]])
b = np.array([[5,6], [7,8]])
c = np.hstack((a,b))
print(c)

# vstack: Stack arrays in sequence vertically (row wise). needs a tuple
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
c = np.vstack((a,b))
print(c)
a = np.array([[1,2], [3,4]])
b = np.array([[5,6], [7,8]])
c = np.vstack((a,b))
print(c)
```

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

```
 [5 6]
 [7 8]]
```

```
# broadcasting
# Broadcasting is a powerful mechanism that allows numpy to work with arrays of
# different shapes when performing arithmetic operations.
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

```
# data science functions and axes
a = np.array([[7,8,9,10,11,12,13], [17,18,19,20,21,22,23]])
print(a)
print(a.sum(axis=None)) # overall sum
print(a.sum())
print(a.sum(axis=0)) # along the rows -> 1 sum entry for each column
print(a.sum(axis=1)) # along the columns -> 1 sum entry for each row

print(a.mean(axis=None)) # overall mean
print(a.mean())
print(a.mean(axis=0)) # along the rows -> 1 mean entry for each column
print(a.mean(axis=1)) # along the columns -> 1 mean entry for each row

# some more: std, var, min, max
```

```
[[ 7  8  9 10 11 12 13]
 [17 18 19 20 21 22 23]]
210
210
[24 26 28 30 32 34 36]
[ 70 140]
15.0
15.0
```

```
      [12. 13. 14. 15. 16. 17. 18.]
      [10. 20.]
```

```
# datatypes
# https://numpy.org/devdocs/user/basics.types.html
# Let numpy choose the datatype
x = np.array([1, 2])
print(x.dtype)

# Let numpy choose the datatype
x = np.array([1.0, 2.0])
print(x.dtype)

# Force a particular datatype, how many bits (how precise)
x = np.array([1, 2], dtype=np.int64) # 8 bytes
print(x.dtype)

x = np.array([1, 2], dtype=np.float32) # 4 bytes
print(x.dtype)
```

```
      int64
      float64
      int64
      float32
```

```
# copy
a = np.array([1,2,3])
b = a # only copies reference!
b[0] = 42
print(a)

a = np.array([1,2,3])
b = a.copy() # actual copy!
b[0] = 42
print(a)
```

```
      [42  2  3]
      [1 2 3]


# generating data
a = np.zeros((2,3)) # size as tuple
b = np.ones((2,3))
print(a)
print(b)
# specific value
c = 5 * np.ones((3,3))
print(c)
c = np.full((3,3),5.0)
print(c)
# identity
d = np.eye(3) #3x3
print(d)

# arange
e = np.arange(10)
print(e)

# linspace
f = np.linspace(0, 10, 5)
print(f)

      [[0. 0. 0.]
       [0. 0. 0.]]
      [[1. 1. 1.]
       [1. 1. 1.]]
      [[5. 5. 5.]
       [5. 5. 5.]
       [5. 5. 5.]]
      [[5. 5. 5.]
       [5. 5. 5.]
       [5. 5. 5.]]
      [[1. 0. 0.]
       [0. 1. 0.]
       [0. 0. 1.]]
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0.   2.5  5.   7.5 10. ]
```

```
# random numbers
a = np.random.random((3,2)) # uniform 0-1 distribution
print(a)
b = np.random.randn(3,2) # normal/Gaussian distribution, mean 0 and unit variance
# no tuple as shape here! each dimension one argument
print(b)

R = np.random.randn(10000)
print(R.mean(), R.var(), R.std())

R = np.random.randn(10, 3)
print(R.mean()) # mean of whole array

# random integer, low,high,size; high is exclusive
R = np.random.randint(3,10,size=(3,3)) # if we only pass one parameter, then from 0-x
print(R)

# with integer is between 0 up to integer exclusive
c = np.random.choice(7, size=10)
print(c)
# with an array it draws random values from this array
d = np.random.choice([1,2,3,4], size=8)
print(d)
```

```
[[0.25797747 0.71255083]
 [0.316169   0.08961938]
 [0.06743722 0.51797696]]
[[-0.43422119 -1.80315952]
 [-1.24003807 -2.28165497]
 [ 0.8919908  -0.6785463 ]]
0.011368774563225052 0.992371465236262 0.9961784304211079
-0.24087934683233125
[[6 7 8]
 [9 7 9]
```

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

```
# eigenvalues
a = np.array([[1,2], [3,4]])
eigenvalues, eigenvectors = np.linalg.eig(a)
# Note: use eigh if your matrix is symmetric (faster)
print(eigenvalues)
print(eigenvectors) # column vectors
print(eigenvectors[:,0]) # column 0 corresponding to eigenvalue[0]

# verify: e-vec * e-val = A * e-vec
d = eigenvectors[:,0] * eigenvalues[0]
e = a @ eigenvectors[:, 0]
print(d, e)
print(d == e) # numerical issues

# correct way to compare matrix
print(np.allclose(d,e))
```

```
[-0.37228132  5.37228132]
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
[-0.82456484  0.56576746]
[ 0.30697009 -0.21062466] [ 0.30697009 -0.21062466]
[ True False]
True
```

```
# solve linear system
# x1+x2=2200
# 1.5 x1 + 4 x2 = 5050
# 2 equations and 2 unknowns
A = np.array([[1, 1], [1.5, 4]])
b = np.array([2200,5050])

# Ax = b <=> x = A-1 b
```

```
# But: inverse is slow and less accurate
x = np.linalg.inv(A).dot(b) # not recommended
print(x)
x = np.linalg.solve(A,b) # good
print(x)
```

```
      [1500.  700.]
      [1500.  700.]
```