

NumPy

Numpy Handbook

By Patrick Loeber - Available at python-engineer.com

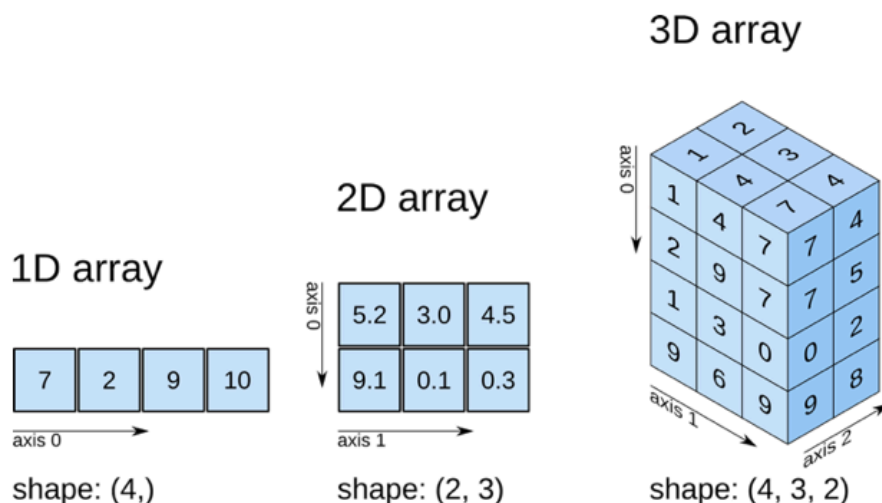
Learn NumPy with this Handbook! It covers code examples for all essential functions and some tricks and useful methods. [NumPy](#) is the core library for scientific computing in Python. It is essential for any **data science** or **machine learning** algorithms.

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, And 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](#)

1. NumPy Introduction

NumPy is the **core library for scientific computing** in Python. The central object in the NumPy library is the NumPy array. The NumPy array is a **high-performance multidimensional array object**, which is designed specifically to perform math operations, **linear algebra**, and probability calculations. Using a NumPy array is usually a lot faster and needs less code than using a Python list. A huge part of the NumPy library consists of C code with the Python API serving as a wrapper around these C functions. This is one of the reasons why NumPy is so fast.



Most of the popular Machine Learning, Deep Learning, and Data Science libraries use NumPy under the hood:

- Scikit-learn
- Matplotlib
- Pandas

Different use cases and operations that can be achieved easily with NumPy:

- Dot product/inner product
- Matrix multiplication
- Element wise matrix product
- Solving linear systems
- Inverse
- Determinant
- Choose random numbers (e.g. Gaussian/Uniform)
- Working with images represented as array
- ... and many more

2. Installation and Array Basics

Installation with **pip** or **Anaconda**:

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

Import numpy:

```
import numpy as np  
# check version  
np.__version__  
# --> 1.19.1
```

Central object is the array:

```
a = np.array([1,2,3,4,5])  
  
a # [1 2 3 4 5]  
a.shape # shape of the array: (5,)  
a.dtype # type of the elements: int32  
a.ndim # number of dimensions: 1  
a.size # total number of elements: 5  
a.itemsize # the size in bytes of each element: 4
```

Essential methods:

```
a = np.array([1,2,3])  
# access and change elements  
print(a[0]) # 1  
a[0] = 5  
print(a) # [5 2 3]  
  
# elementwise math operations  
b = a * np.array([2,0,2])  
print(b) # [10 0 6]  
print(a.sum()) # 10
```

3. Array vs List

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

# 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) # [1, 2, 3, 4, 5]

a2 = a + np.array([4])
print(a2) # this is called broadcasting, adds 4 to each element
# -> [5 6 7]

# vector addition (this is technically correct compared to broadcasting)
a3 = a + np.array([4,4,4])
print(a3) # [5 6 7]

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

# multiplication
l2 = 2 * l # list l repeated 2 times, same as l+l
print(l2)
# -> [1, 2, 3, 4, 1, 2, 3, 4]

a3 = 2 * a # multiplication for each element

print(a3)
# -> [2 4 6]

# modify each item in the list
l2 = []
for i in l:
    l2.append(i**2)
print(l2) # [1, 4, 9, 16]

# or list comprehension
l2 = [i**2 for i in l]
print(l2) # [1, 4, 9, 16]

a2 = a**2 # -> squares each element!
```

```
print(a2) # [1 4 9]

# Note: function applied to array usually operates element wise
a2 = np.sqrt(a) # np.exp(a), np.tanh(a)
print(a2) # [1. 1.41421356 1.73205081]
a2 = np.log(a)
print(a2) # [0. 0.69314718 1.09861229]
```

4. Dot 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) # 11

# easy with numpy :)
dot = np.dot(a,b)
print(dot) # 11

# step by step manually
c = a * b
print(c) # [3 8]
d = np.sum(c)
print(d) # 11

# most of these functions are also instance methods
dot = a.dot(b)
print(dot) # 11
dot = (a*b).sum()
print(dot) # 11

# in newer versions
dot = a @ b
print(dot) # 11
```

5. Speed Test Array vs List

```
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) # -> 0.19371
print('Time with array:', t2) # -> 0.00112
print('Ratio', t1/t2)         # -> 172.332 times faster
```

6. Multidimensional (nd) Arrays

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

```

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

print(a.shape) # (2, 2)

# Access elements
# row first, then columns

print(a[0]) # [1 2]
print(a[0][0]) # 1
# or
print(a[0,0]) # 1

# slicing
print(a[:,0]) # all rows in col 0: [1 3]
print(a[0,:]) # all columns in row 0: [1 2]

# transpose
a.T

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

d = a * b # elementwise multiplication

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

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

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

# diag
c = np.diag(a)
print(c) # [1 4]

# diag on a vector returns diagonal matrix (overloaded function)
c = np.diag([1,4])
print(c)
# [[1 0]
#  [0 4]]

```

7. Indexing, Slicing, And Boolean Indexing

Indexing and Slicing:

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

# Integer array indexing
b = a[0,1]
print(b) # 2

# Slicing
row0 = a[0,:]
print(row0) # [1 2 3 4]

col0 = a[:, 0]
print(col0) # [1 5 9]

slice_a = a[0:2,1:3]
print(slice_a)
# [[2 3]
#   [6 7]]

# indexing starting from the end: -1, -2 etc...
last = a[-1,-1]
print(last) # 12
```

Boolean indexing:

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

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



```

# [[False False]
#  [ True True]
#  [ True True]]

# note: this will be a rank 1 array!
print(a[bool_idx]) # [3 4 5 6]

# We can do all of the above in a single concise statement:
print(a[a > 2]) # [3 4 5 6]

# np.where(): same size with modified values
b = np.where(a>2, a, -1)
print(b)
# [[-1 -1]
#  [ 3 4]
#  [ 5 6]]

# fancy indexing: access multiple indices at once
a = np.array([10,19,30,41,50,61])

b = a[[1,3,5]]
print(b) # [19 41 61]

# compute indices where condition is True
even = np.argwhere(a%2==0).flatten()
print(even) # [0 2 4]

a_even = a[even]
print(a_even) # [10 30 50]

```

8. Reshaping

```

a = np.arange(1, 7)
print(a) # [1 2 3 4 5 6]

b = a.reshape((2, 3)) # error if shape cannot be used
print(b)
# [[1 2 3]
#  [4 5 6]]

c = a.reshape((3, 2)) # 3 rows, 2 columns
print(c)
# [[1 2]
#  [3 4]
#  [5 6]]

```

```

# 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) # (6,)

d = a[np.newaxis, :]
print(d) # [[1 2 3 4 5 6]]
print(d.shape) # (1, 6)

e = a[:, np.newaxis]
print(e)
# [[1]
#  [2]
#  [3]
#  [4]
#  [5]
#  [6]]

print(e.shape) # (6, 1)

```

9. Concatenation

```

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

# combine into 1d
c = np.concatenate((a, b), axis=None)
print(c) # [1 2 3 4 5 6]

# add new row
d = np.concatenate((a, b), axis=0)
print(d)
# [[1 2]
#  [3 4]
#  [5 6]]

# add new column: note that we have to transpose b!
e = np.concatenate((a, b.T), axis=1)
print(e)
# [[1 2 5]
#  [3 4 6]]

# 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) # [1 2 3 4 5 6 7 8]

a = np.array([[1,2], [3,4]])
b = np.array([[5,6], [7,8]])
c = np.hstack((a,b))
print(c)
# [[1 2 5 6]
#   [3 4 7 8]]

# 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)
# [[1 2 3 4]
#   [5 6 7 8]]

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]
#   [7 8]]

```

10. Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

```

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
y = np.array([1, 0, 1])
z = x + y # Add v to each row of x using broadcasting
print(z)
# [[ 2  2  4]
#   [ 5  5  7]
#   [ 8  8 10]
#   [11 11 13]]

```

11. Functions and Axis

```
a = np.array([[7,8,9,10,11,12,13], [17,18,19,20,21,22,23]])

print(a.sum())          # default=None-> 210
print(a.sum(axis=None)) # overall sum -> 210

print(a.sum(axis=0)) # along the rows -> 1 sum entry for each column
# -> [24 26 28 30 32 34 36]

print(a.sum(axis=1)) # along the columns -> 1 sum entry for each row
# -> [ 70 140]

print(a.mean())          # default=None-> 15.0
print(a.mean(axis=None)) # overall mean -> 15.0

print(a.mean(axis=0)) # along the rows -> 1 mean entry for each column
# -> [12. 13. 14. 15. 16. 17. 18.]

print(a.mean(axis=1)) # along the columns -> 1 mean entry for each row
# -> [10. 20.]

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

12. Datatypes

[Overview of all datatypes](#)

```
# Let numpy choose the datatype
x = np.array([1, 2])
print(x.dtype) # int32

# Let numpy choose the datatype
x = np.array([1.0, 2.0])
print(x.dtype) # float64

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

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

13. Copying

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

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

14. Generating Arrays

```
# zeros
a = np.zeros((2,3)) # size as tuple
# [[0. 0. 0.]
#  [0. 0. 0.]]

# ones
b = np.ones((2,3))
# [[1. 1. 1.]
#  [1. 1. 1.]]

# specific value
c = np.full((3,3),5.0)
# [[5. 5. 5.]
#  [5. 5. 5.]
#  [5. 5. 5.]]

# identity
d = np.eye(3) #3x3
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]

# arange
e = np.arange(10)
# [0 1 2 3 4 5 6 7 8 9]

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

```
# [ 0. 2.5 5. 7.5 10. ]
```

15. Random Numbers

```
a = np.random.random((3,2)) # uniform 0-1 distribution
# [[0.06121857 0.10180167]
#  [0.83321726 0.54906613]
#  [0.94170273 0.19447411]]

b = np.random.randn(3,2) # normal/Gaussian distribution, mean 0 and unit
variance
# no tuple as shape here! each dimension one argument
# [[ 0.56759123 -0.65068333]
#  [ 0.83445762 -0.36436185]
#  [ 1.27150812 -0.32906051]]

c = np.random.randn(10000)
print(c.mean(), c.var(), c.std())
# -0.0014 0.9933 0.9966

d = np.random.randn(10, 3)
print(d.mean()) # mean of whole array: -0.1076827228882305

# random integer, low,high,size; high is exclusive
e = np.random.randint(3,10,size=(3,3)) # if we only pass one parameter, then
from 0-x
print(e)
# [[6 8 4]
#  [3 6 3]
#  [4 7 8]]

# with integer is between 0 up to integer exclusive
f = np.random.choice(7, size=10)
# [2 0 4 5 1 3 4 0 0 6]

# with an array it draws random values from this array
g = np.random.choice([1,2,3,4], size=8)
# [4 2 1 3 4 1 4 1]
```

16. Linear Algebra (Eigenvalues / Solving Linear Systems)

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)
# [-0.37228132  5.37228132]

print(eigenvectors) # column vectors
# [[-0.82456484 -0.41597356]
#   [ 0.56576746 -0.90937671]]

print(eigenvectors[:,0]) # column 0 corresponding to eigenvalue[0]
# [-0.82456484  0.56576746]

# verify: e-vec * e-val = A * e-vec
d = eigenvectors[:,0] * eigenvalues[0]
e = a @ eigenvectors[:, 0]

print(d, e) # [ 0.30697009 -0.21062466] [ 0.30697009 -0.21062466]
# looks the same, but:
print(d == e) # [ True False] -> numerical issues

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

Solving Linear Systems

```
#      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) # [1500. 700.]
```

```
# instead use:  
x = np.linalg.solve(A,b) # good  
print(x) # [1500. 700.]
```

17. Loading CSV files

```
# 1) load with np.loadtxt()  
# skiprows=1, ...  
data = np.loadtxt('my_file.csv', delimiter=",", dtype=np.float32)  
print(data.shape, data.dtype)  
  
# 2) load with np.genfromtxt()  
# similar but slightly more configuration parameters  
# skip_header=0, missing_values="---", filling_values=0.0, ...  
data = np.genfromtxt('my_file.csv', delimiter=",", dtype=np.float32)  
print(data.shape)
```