**Important things you should know about Numpy and Pandas**

1. The data manipulation capabilities of pandas are built on top of the numpy library. In a way, numpy is a dependency of the **pandas** library.
2. **Pandas is best at handling tabular data sets comprising different variable types (integer, float, double, etc.). In addition, the pandas library can also be used to perform even the most naive of tasks such as loading data or doing feature engineering on time series data.**
3. **Numpy is most suitable for performing basic numerical computations such as mean, median, range, etc. Alongside, it also supports the creation of multi-dimensional arrays.**
4. **Numpy library can also be used to integrate C/C++ and Fortran code.**
5. **Remember, python is a zero indexing language unlike R where indexing starts at one.**
6. The best part of learning pandas and numpy is the strong active community support you'll get from around the world.

Just to give you a flavor of the numpy library, we'll quickly go through its syntax structures and some important commands such as slicing, indexing, concatenation, etc. All these commands will come in handy when using pandas as well. Let's get started!

**Starting with Numpy**

#load the library and check its version, just to make sure we aren't using an older version

import numpy as np

np.\_\_version\_\_

#create a list comprising numbers from 0 to 9

L = list(range(10))

>>> print(L)

output

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

#converting integers to string - this style of handling lists is known as list comprehension.

**#List comprehension offers a versatile way to handle list #manipulations tasks easily.**

Here's an example.

[str(c) for c in L]

OP:

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

[type(item) for item in L]

[int, int, int, int, int, int, int, int, int, int]

**Creating Arrays**

# Array creation

## Introduction

There are 5 general mechanisms for creating arrays:

1. **Conversion from other Python structures (e.g., lists, tuples)**
2. **Intrinsic numpy array creation objects (e.g., arange, ones, zeros, etc.)**
3. **Reading arrays from disk, either from standard or custom formats**
4. **Creating arrays from raw bytes through the use of strings or buffers**
5. **Use of special library functions (e.g., random)**

This section will not cover means of replicating, joining, or otherwise expanding or mutating existing arrays. Nor will it cover creating object arrays or structured arrays. Both of those are covered in their own sections.

## Converting Python array\_like Objects to NumPy Arrays

**In general, numerical data arranged in an array-like structure in Python can be converted to arrays through the use of the array() function. The most obvious examples are lists and tuples.**

Some objects may support the array-protocol and allow conversion to arrays this way. **A simple way to find out if the object can be converted to a numpy array using array()** is simply to try it interactively and see if it works! (The Python Way).

Examples:

>>>

**>>>** x = np.array([2,3,1,0])

**>>>** x = np.array([2, 3, 1, 0])

**>>>** x = np.array([[1,2.0],[0,0]])

**>>>** x = np.array([[1,2.0],[0,0],(1+1j,3.)])

*# note mix of tuple and lists,* and types

**>>> x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j], [ 1.+1.j, 3.+0.j]])**

## Intrinsic NumPy Array Creation

NumPy has built-in functions for creating arrays from scratch:

**zeros(shape) will create an array filled with 0 values with the specified shape. The default dtype is float64.**

>>>

**>>>** np.zeros((2, 3))

array([[ 0., 0., 0.], [ 0., 0., 0.]])

ones(shape) will create an array filled with 1 values. It is identical to zeros in all other respects.

**arange() will create arrays with regularly incrementing values.** Check the doc string for complete information on the various ways it can be used. A few examples will be given here:

>>>

**>>>** np.arange(10)

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

**>>>** np.arange(2, 10, dtype=float)

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

**>>>** np.arange(2, 3, 0.1)

array([ 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])

Note that there are some subtleties regarding the last usage that the user should be aware of that are described in the arange docstring.

**linspace() will create arrays with a specified number of elements, and spaced equally between the specified beginning and end values. For example:**

>>>

**>>> np.linspace(1., 4., 6)**

**array([ 1. , 1.6, 2.2, 2.8, 3.4, 4. ])**

The advantage of this creation function is that one can guarantee the number of elements and the starting and end point, **which arange() generally will not do for arbitrary start, stop, and step values.**

**indices() will create a set of arrays (stacked as a one-higher dimensioned array), one per dimension with each representing variation in that dimension.** An example illustrates much better than a verbal description:

>>>

**>>> np.indices((3,3))**

array([[[0, 0, 0], [1, 1, 1], [2, 2, 2]], [[0, 1, 2], [0, 1, 2], [0, 1, 2]]])

**This is particularly useful for evaluating functions of multiple dimensions on a regular grid.**

**Numpy arrays are homogeneous in nature, i.e., they comprise one data type (integer, float, double, etc.) unlike lists.**

#creating arrays

np.zeros(10, dtype='int')

**Output:**

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

**#creating a 3 row x 5 column matrix**

np.ones((3,5), dtype=float)

array([[ 1., 1., 1., 1., 1.],

[ 1., 1., 1., 1., 1.],

[ 1., 1., 1., 1., 1.]])

**#creating a matrix with a predefined value**

**np.full((3,5),1.23)**

**output:**

array([[ 1.23, 1.23, 1.23, 1.23, 1.23],

[ 1.23, 1.23, 1.23, 1.23, 1.23],

[ 1.23, 1.23, 1.23, 1.23, 1.23]])

**#create an array with a set sequence**

**np.arange(0, 20, 2)**

array([0, 2, 4, 6, 8,10,12,14,16,18])

**#create an array of even space between the given range of values**

**np.linspace(0, 2, 5)**

**output:**

array([ 0., 0.25, 0.5 , 0.75, 1.])

**#create a 3x3 array with mean 0 and standard #deviation 1 in a given dimension**

**np.random.normal(0, 1, (3,3))**

**output:**

array([[ 0.72432142, -0.90024075, 0.27363808],

[ 0.88426129, 1.45096856, -1.03547109],

[-0.42930994, -1.02284441, -1.59753603]])

**#create an identity matrix**

**np.eye(3)**

**output**

array([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]])

**#set a random seed**

np.random.seed(0)

x1 = np.random.randint(10, size=6) #one dimension

x2 = np.random.randint(10, size=(3,4)) #two dimension

x3 = np.random.randint(10, size=(3,4,5)) #three dimension

print("x3 ndim:", x3.ndim)

print("x3 shape:", x3.shape)

print("x3 size: ", x3.size)

**Output:**

('x3 ndim:', 3)

('x3 shape:', (3, 4, 5))

('x3 size: ', 60)

**Array Indexing**

The important thing to remember is that indexing in python starts at zero.

x1 = np.array([4, 3, 4, 4, 8, 4])

print(x1)

**output:**

array([4, 3, 4, 4, 8, 4])

**#assess value to index zero**

x1[0]

**output**

4

**#assess fifth value**

x1[4]

**output**

8

**#get the last value**

x1[-1]

**output**

4

**#get the second last value**

x1[-2]

**output**

8

**#in a multidimensional array, we need to specify row and #column index (3x4) matrix**

x2=np.array([[3, 7, 5, 5],

[0, 1, 5, 9],

[3, 0, 5, 0]])

**#1st row and 2nd column value**

**x2[2,3]**

0

**#3rd row and last value from the 3rd column**

x2[2,-1]

0

**#replace value at 0,0 index**

x2[0,0] = 12

x2

**output:**

array([[12, 7, 5, 5],

[ 0, 1, 5, 9],

[ 3, 0, 5, 0]])

**Array Slicing**

**Now, we'll learn to access multiple or a range of elements from an array.**

**x = np.arange(10)**

x

**output:**

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

**#from start 0 to 5th position**

x[:5]

**output:**

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

**#from 4th position to end**

x[4:]

**output:**

array([4, 5, 6, 7, 8, 9])

**#from 4th to 6th position**

x[4:7]

**output**

array([4, 5, 6])

**#return elements at even place**

x[ : : 2]

**output:**

array([0, 2, 4, 6, 8])

**#return elements from first position step by two**

x[1::2]

**output:**

array([1, 3, 5, 7, 9])

**#reverse the array**

x[::-1]

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

**Array Concatenation**

Many a time, we are required to combine different arrays. So, instead of typing each of their elements manually, you can use array concatenation to handle such tasks easily.

**#You can concatenate two or more arrays at once.**

**x = np.array([1, 2, 3])**

**y = np.array([3, 2, 1])**

**z = [21,21,21]**

**np.concatenate([x, y,z])**

**output:**

array([ 1, 2, 3, 3, 2, 1, 21, 21, 21])

**#You can also use this function to create 2-dimensional #arrays.**

**grid = np.array([[1,2,3],[4,5,6]])**

**np.concatenate([grid,grid])**

**Output:**

array([[1, 2, 3],

[4, 5, 6],

[1, 2, 3],

[4, 5, 6]])

**#Using its axis parameter, you can define row-wise or #column-wise matrix**

**np.concatenate([grid,grid],axis=1)**

**output**

array([[1, 2, 3, 1, 2, 3],

[4, 5, 6, 4, 5, 6]])

Until now, we used the concatenation function of arrays of equal dimension. But, what if you are required **to combine a 2D array with 1D array? In such situations, np.concatenate might not be the best option to use. Instead, you can use np.vstack or np.hstack to do the task. Let's see how!**

**x = np.array([3,4,5])**

**grid = np.array([[1,2,3],[17,18,19]])**

**np.vstack([x,grid])**

**output:**

array([[ 3, 4, 5],

[ 1, 2, 3],

[17, 18, 19]])

**#Similarly, you can add an array using np.hstack**

**z = np.array([[9],[9]])**

**np.hstack([grid,z])**

**output**

array([[ 1, 2, 3, 9],

[17, 18, 19, 9]])

**Also, we can split the arrays based on pre-defined positions. Let's see how!**

x = np.arange(10)

x

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

**If *indices\_or\_sections* is a 1-D array of sorted integers, the entries indicate where along *axis*the array is split. For example, [2, 3] would, for axis=0, result in**

* **ary[:2]**
* **ary[2:3]**
* **ary[3:]**

**If an index exceeds the dimension of the array along *axis*, an empty sub-array is returned correspondingly.**

x1,x2,x3 = np.split(x,[3,6])

print (x1,x2,x3)

**output**

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

**grid = np.arange(16).reshape((4,4))**

**grid**

**upper,lower = np.vsplit(grid,[2])**

**print (upper, lower)**

(array([[0, 1, 2, 3],

[4, 5, 6, 7]]), array([[ 8, 9, 10, 11],

[12, 13, 14, 15]]))

In addition to the functions we learned above, there are several other mathematical functions available in the numpy library such as sum, divide, multiple, abs, power, mod, sin, cos, tan, log, var, min, mean, max, etc. which you can be used to perform basic arithmetic calculations.

## How To Visualize NumPy Arrays

Lastly, something that will definitely come in handy is to know how you can plot your arrays. This can especially be handy in data exploration, but also in later stages of the data science workflow, when you want to visualize your arrays.

### With np.histogram() Contrary to what the function might suggest, the np.histogram() function doesn’t draw the histogram but it does compute the occurrences of the array that fall within each bin; This will determine the area that each bar of your histogram takes up.

What you pass to the np.histogram() function then is first the input data or the array that you’re working with. The array will be flattened when the histogram is computed.

# Import `numpy` as `np`

import numpy as np

# Initialize your array

my\_3d\_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype=np.int64)

# Pass the array to `np.histogram()`

print(np.histogram(my\_3d\_array))

**# Specify the number of bins**

**print(np.histogram(my\_3d\_array, bins=range(0,13)))**

Output:

(array([4, 2, 2, 1, 1, 1, 1, 1, 1, 2]), array([ 1. , 2.1, 3.2, 4.3, 5.4, 6.5, 7.6, 8.7, 9.8,

10.9, 12. ]))

(array([0, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2]), array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]))

**Visualization is a piece of cake with the help of Matplotlib, but you don’t need np.histogram() to compute the histogram. plt.hist() does this for itself when you pass it the (flattened) data and the bins:**

**# Import numpy and matplotlib**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Construct the histogram with a flattened 3d array and a range #of bins**

**plt.hist(my\_3d\_array.ravel(), bins=range(0,13))**

**# Add a title to the plot**

**plt.title('Frequency of My 3D Array Elements')**

**# Show the plot**

**plt.show()**