Numpy Tutorial – Introduction to ndarray

This is part 1 of the numpy tutorial covering all the core aspects of performing data manipulation and analysis with numpy's ndarrays. Numpy is the most basic and a powerful package for scientific computing and data manipulation in python.

Numpy Tutorial Part 1: Introduction to Arrays. Photo by Bryce Canyon.

Contents

1. Introduction to Numpy

Numpy is the most basic and a powerful package for working with data in python.

If you are going to work on data analysis or machine learning projects, then having a solid understanding of numpy is nearly mandatory.

Because other packages for data analysis (like pandas) is built on top of numpy and the scikit-learn package which is used to build machine learning applications works heavily with numpy as well.

So what does numpy provide?

At the core, numpy provides the excellent ndarray objects, short for n-dimensional arrays.

In a 'ndarray' object, aka 'array', you can store multiple items of the same data type. It is the facilities around the array object that makes numpy so convenient for performing math and data manipulations.

You might wonder, *'I can store numbers and other objects in a python list itself and do all sorts of computations and manipulations through list comprehensions, for-loops etc. What do I need a numpy array for?'*

Well, there are very significant advantages of using numpy arrays overs lists.

To understand this, let's first see how to create a numpy array.

2. How to create a numpy array?

There are multiple ways to create a numpy array, most of which will be covered as you read this. However one of the most common ways is to create one from a list or a list like an object by passing it to the `np.array` function.

```
# Create an 1d array from a list
import numpy as np
list1 = [0,1,2,3,4]
arr1d = np.array(list1)

# Print the array and its type
print(type(arr1d))
arr1d

#> class 'numpy.ndarray'
#> array([0, 1, 2, 3, 4])
```

The key difference between an array and a list is, arrays are designed to handle vectorized operations while a python list is not.

That means, if you apply a function it is performed on every item in the array, rather than on the whole array object.

Let's suppose you want to add the number 2 to every item in the list. The intuitive way to do it is something like this:

```
list1 + 2  # error
```

That was not possible with a list. But you can do that on a ndarray.

```
# Add 2 to each element of arr1d
arr1d + 2

#> array([2, 3, 4, 5, 6])
```

Another characteristic is that, once a numpy array is created, you cannot increase its size. To do so, you will have to create a new array. But such a behavior of extending the size is natural in a list.

Nevertheless, there are so many more advantages. Let's find out.

So, that's about 1d array. You can also pass a list of lists to create a matrix like a 2d array.

```
# Create a 2d array from a list of lists
list2 = [[0,1,2], [3,4,5], [6,7,8]]
arr2d = np.array(list2)
arr2d

#> array([[0, 1, 2],
#>        [3, 4, 5],
#>        [6, 7, 8]])
```

You may also specify the datatype by setting the dtype argument. Some of the most commonly used numpy dtypes are: `'float'`, `'int'`, `'bool'`, `'str'` and `'object'`.

To control the memory allocations you may choose to use one of 'float32', 'float64', 'int8', 'int16' or 'int32'.

```
# Create a float 2d array
arr2d_f = np.array(list2, dtype='float')
arr2d_f

#> array([[ 0.,  1.,  2.],
#>        [ 3.,  4.,  5.],
#>        [ 6.,  7.,  8.]])
```

The decimal point after each number is indicative of the float datatype. You can also convert it to a different datatype using the `astype` method.

```
# Convert to 'int' datatype
arr2d_f.astype('int')

#> array([[0, 1, 2],
#>        [3, 4, 5],
#>        [6, 7, 8]])
```

```
# Convert to int then to str datatype
arr2d_f.astype('int').astype('str')

#> array([['0', '1', '2'],
#>        ['3', '4', '5'],
#>        ['6', '7', '8']],
#>       dtype='U21')
```

A numpy array must have all items to be of the same data type, unlike lists. This is another significant difference.

However, if you are uncertain about what datatype your array will hold or if you want to hold characters and numbers in the same array, you can set the `dtype` as `'object'`.

```
# Create a boolean array
arr2d_b = np.array([1, 0, 10], dtype='bool')
arr2d_b

#> array([ True, False,  True], dtype=bool)
```

```
# Create an object array to hold numbers as well as strings
arr1d_obj = np.array([1, 'a'], dtype='object')
arr1d_obj

#> array([1, 'a'], dtype=object)
```

Finally, you can always convert an array back to a python list using `tolist()`.

```
# Convert an array back to a list
arr1d_obj.tolist()

#> [1, 'a']
```

To summarise, the main differences with python lists are:

1. Arrays support vectorised operations, while lists don't.
2. Once an array is created, you cannot change its size. You will have to create a new array or overwrite the existing one.
3. Every array has one and only one dtype. All items in it should be of that dtype.
4. An equivalent numpy array occupies much less space than a python list of lists.

3. How to inspect the size and shape of a numpy array?

Every array has some properties I want to understand in order to know about the array.

Let's consider the array, arr2d. Since it was created from a list of lists, it has 2 dimensions that can be shown as rows and columns, like in a matrix.

Had I created one from a list of list of lists, it would have 3 dimensions, as in a cube. And so on.

Let's suppose you were handed a numpy vector that you didn't create yourself. What are the things you would want to explore in order to know about that array?

Well, I want to know:

```
<li>If it is a 1D or a 2D array or more. (ndim)</li>

<li>How many items are present in each dimension (shape)</li>

<li>What is its datatype (dtype)</li>

<li>What is the total number of items in it (size)</li>

<li>Samples of first few items in the array (through indexing)</li>
```

```
# Create a 2d array with 3 rows and 4 columns
list2 = [[1, 2, 3, 4],[3, 4, 5, 6], [5, 6, 7, 8]]
arr2 = np.array(list2, dtype='float')
arr2

#> array([[ 1.,  2.,  3.,  4.],
#>        [ 3.,  4.,  5.,  6.],
#>        [ 5.,  6.,  7.,  8.]])
```

```
# shape
print('Shape: ', arr2.shape)

# dtype
print('Datatype: ', arr2.dtype)

# size
print('Size: ', arr2.size)

# ndim
print('Num Dimensions: ', arr2.ndim)

#> Shape:  (3, 4)
#> Datatype:  float64
#> Size:  12
#> Num Dimensions:  2
```

4. How to extract specific items from an array?

```
arr2

#> array([[ 1.,  2.,  3.,  4.],
#>        [ 3.,  4.,  5.,  6.],
#>        [ 5.,  6.,  7.,  8.]])
```

You can extract specific portions on an array using indexing starting with 0, something similar to how you would do with python lists.

But unlike lists, numpy arrays can optionally accept as many parameters in the square brackets as there is number of dimensions.

```
# Extract the first 2 rows and columns
arr2[:2, :2]
list2[:2, :2]  # error

#> array([[ 1.,  2.],
#>        [ 3.,  4.]])
```

Additionally, numpy arrays support boolean indexing.

A boolean index array is of the same shape as the array-to-be-filtered and it contains only True and False values. The values corresponding to True positions are retained in the output.

```
# Get the boolean output by applying the condition to each element.
b = arr2 > 4
b

#> array([[False, False, False, False],
#>        [False, False,  True,  True],
#>        [ True,  True,  True,  True]], dtype=bool)
```

```
arr2[b]

#> array([ 5.,  6.,  5.,  6.,  7.,  8.])
```

4.1 How to reverse the rows and the whole array?

Reversing an array works like how you would do with lists, but you need to do for all the axes (dimensions) if you want a complete reversal.

```
# Reverse only the row positions
arr2[::-1, ]

#> array([[ 5.,  6.,  7.,  8.],
#>        [ 3.,  4.,  5.,  6.],
#>        [ 1.,  2.,  3.,  4.]])
```

```
# Reverse the row and column positions
arr2[::-1, ::-1]

#> array([[ 8.,  7.,  6.,  5.],
#>        [ 6.,  5.,  4.,  3.],
#>        [ 4.,  3.,  2.,  1.]])
```

4.2 How to represent missing values and infinite?

Missing values can be represented using `np.nan` object, while np.inf represents infinite. Let's place some in arr2d.

```
# Insert a nan and an inf
arr2[1,1] = np.nan  # not a number
arr2[1,2] = np.inf  # infinite
arr2

#> array([[  1.,   2.,   3.,   4.],
#>        [  3.,  nan,  inf,   6.],
#>        [  5.,   6.,   7.,   8.]])
```

```
# Replace nan and inf with -1. Don't use arr2 == np.nan
missing_bool = np.isnan(arr2) | np.isinf(arr2)
arr2[missing_bool] = -1
arr2

#> array([[ 1.,  2.,  3.,  4.],
#>        [ 3., -1., -1.,  6.],
#>        [ 5.,  6.,  7.,  8.]])
```

4.3 How to compute mean, min, max on the ndarray?

The ndarray has the respective methods to compute this for the whole array.

```
# mean, max and min
print("Mean value is: ", arr2.mean())
print("Max value is: ", arr2.max())
print("Min value is: ", arr2.min())

#> Mean value is:  3.58333333333
#> Max value is:  8.0
#> Min value is:  -1.0
```

However, if you want to compute the minimum values row wise or column wise, use the np.amin version instead.

```
# Row wise and column wise min
print("Column wise minimum: ", np.amin(arr2, axis=0))
print("Row wise minimum: ", np.amin(arr2, axis=1))

#> Column wise minimum:  [ 1. -1. -1.  4.]
#> Row wise minimum:  [ 1. -1.  5.]
```

Computing the minimum row-wise is fine. But what if you want to do some other computation/function row-wise? It can be done using the `np.apply_over_axis` which you will see in the upcoming topic.

```
# Cumulative Sum
np.cumsum(arr2)

#> array([  1.,   3.,   6.,  10.,  13.,  12.,  11.,  17.,  22.,  28.,  35., 43.])
```

5. How to create a new array from an existing array?

If you just assign a portion of an array to another array, the new array you just created actually refers to the parent array in memory.

That means, if you make any changes to the new array, it will reflect in the parent array as well.

So to avoid disturbing the parent array, you need to make a copy of it using copy(). All numpy arrays come with the copy() method.

```
# Assign portion of arr2 to arr2a. Doesn't really create a new array.
arr2a = arr2[:2,:2]
arr2a[:1, :1] = 100  # 100 will reflect in arr2
arr2

#> array([[ 100.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
```

```
# Copy portion of arr2 to arr2b
arr2b = arr2[:2, :2].copy()
arr2b[:1, :1] = 101  # 101 will not reflect in arr2
arr2

#> array([[ 100.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
```

6. Reshaping and Flattening Multidimensional arrays

Reshaping is changing the arrangement of items so that shape of the array changes while maintaining the same number of dimensions.

Flattening, however, will convert a multi-dimensional array to a flat 1d array. And not any other shape.

First, let's reshape the `arr2` array from 3×4 to 4×3 shape.

```
# Reshape a 3x4 array to 4x3 array
arr2.reshape(4, 3)

#> array([[ 100.,    2.,    3.],
#>        [   4.,    3.,   -1.],
#>        [  -1.,    6.,    5.],
#>        [   6.,    7.,    8.]])
```

6.1 What is the difference between flatten() and ravel()?

There are 2 popular ways to implement flattening. That is using the `flatten()` method and the other using the `ravel()` method.

The difference between ravel and flatten is, the new array created using ravel is actually a reference to the parent array. So, any changes to the new array will affect the parent as well. But is memory efficient since it does not create a copy.

```
# Flatten it to a 1d array
arr2.flatten()

#> array([ 100.,    2.,    3.,    4.,    3.,   -1.,   -1.,    6.,    5., 6.,    7.,
8.])
```

```
# Changing the flattened array does not change parent
b1 = arr2.flatten()
b1[0] = 100  # changing b1 does not affect arr2
arr2

#> array([[ 100.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
```

```
# Changing the raveled array changes the parent also.
b2 = arr2.ravel()
b2[0] = 101  # changing b2 changes arr2 also
arr2

#> array([[ 101.,    2.,    3.,    4.],
#>        [   3.,   -1.,   -1.,    6.],
#>        [   5.,    6.,    7.,    8.]])
```

7. How to create sequences, repetitions and random numbers using numpy?

The `np.arange` function comes handy to create customised number sequences as `ndarray`.

```
# Lower limit is 0 be default
print(np.arange(5))

# 0 to 9
print(np.arange(0, 10))

# 0 to 9 with step of 2
print(np.arange(0, 10, 2))

# 10 to 1, decreasing order
print(np.arange(10, 0, -1))

#> [0 1 2 3 4]
#> [0 1 2 3 4 5 6 7 8 9]
#> [0 2 4 6 8]
#> [10  9  8  7  6  5  4  3  2  1]
```

You can set the starting and end positions using np.arange. But if you are focussed on the number of items in the array you will have to manually calculate the appropriate step value.

Say, you want to create an array of exactly 10 numbers between 1 and 50, Can you compute what would be the step value?

Well, I am going to use the `np.linspace` instead.

```
# Start at 1 and end at 50
np.linspace(start=1, stop=50, num=10, dtype=int)

#> array([ 1,  6, 11, 17, 22, 28, 33, 39, 44, 50])
```

Notice since I explicitly forced the dtype to be int, the numbers are not equally spaced because of the rounding.

Similar to np.linspace, there is also np.logspace which rises in a logarithmic scale. In np.logspace, the given start value is actually base^start and ends with base^stop, with a default based value of 10.

```
# Limit the number of digits after the decimal to 2
np.set_printoptions(precision=2)

# Start at 10^1 and end at 10^50
np.logspace(start=1, stop=50, num=10, base=10)

#> array([  1.00e+01,   2.78e+06,   7.74e+11,   2.15e+17,   5.99e+22,
#>          1.67e+28,   4.64e+33,   1.29e+39,   3.59e+44,   1.00e+50])
```

The `np.zeros` and `np.ones` functions lets you create arrays of desired shape where all the items are either 0's or 1's.

```
np.zeros([2,2])
#> array([[ 0.,  0.],
#>        [ 0.,  0.]])
```

```
np.ones([2,2])
#> array([[ 1.,  1.],
#>        [ 1.,  1.]])
```

7.1 How to create repeating sequences?

`np.tile` will repeat a whole list or array n times. Whereas, `np.repeat` repeats each item n times.

```
a = [1,2,3]

# Repeat whole of 'a' two times
print('Tile:   ', np.tile(a, 2))

# Repeat each element of 'a' two times
print('Repeat: ', np.repeat(a, 2))

#> Tile:    [1 2 3 1 2 3]
#> Repeat:  [1 1 2 2 3 3]
```

7.2 How to generate random numbers?

The `random` module provides nice functions to generate random numbers (and also statistical distributions) of any given shape.

```
# Random numbers between [0,1) of shape 2,2
print(np.random.rand(2,2))

# Normal distribution with mean=0 and variance=1 of shape 2,2
print(np.random.randn(2,2))

# Random integers between [0, 10) of shape 2,2
print(np.random.randint(0, 10, size=[2,2]))

# One random number between [0,1)
print(np.random.random())

# Random numbers between [0,1) of shape 2,2
print(np.random.random(size=[2,2]))

# Pick 10 items from a given list, with equal probability
print(np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10))

# Pick 10 items from a given list with a predefined probability 'p'
print(np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10, p=[0.3, .1, 0.1, 0.4, 0.
1]))  # picks more o's

#> [[ 0.84  0.7 ]
#>  [ 0.52  0.8 ]]

#> [[-0.06 -1.55]
#>  [ 0.47 -0.04]]

#> [[4 0]
#>  [8 7]]

#> 0.08737272424956832

#> [[ 0.45  0.78]
#>  [ 0.03  0.74]]

#> ['i' 'a' 'e' 'e' 'a' 'u' 'o' 'e' 'i' 'u']
#> ['o' 'a' 'e' 'a' 'a' 'o' 'o' 'o' 'a' 'o']
```

Now, everytime you run any of the above functions, you get a different set of random numbers.

If you want to repeat the same set of random numbers every time, you need to set the seed or the random state. The see can be any value. The only requirement is you must set the seed to the same value every time you want to generate the same set of random numbers.

Once `np.random.RandomState` is created, all the functions of the `np.random` module becomes available to the created randomstate object.

```
# Create the random state
rn = np.random.RandomState(100)

# Create random numbers between [0,1) of shape 2,2
print(rn.rand(2,2))

#> [[ 0.54  0.28]
#>  [ 0.42  0.84]]
```

```
# Set the random seed
np.random.seed(100)

# Create random numbers between [0,1) of shape 2,2
print(np.random.rand(2,2))

#> [[ 0.54  0.28]
#>  [ 0.42  0.84]]
```

7.3 How to get the unique items and the counts?

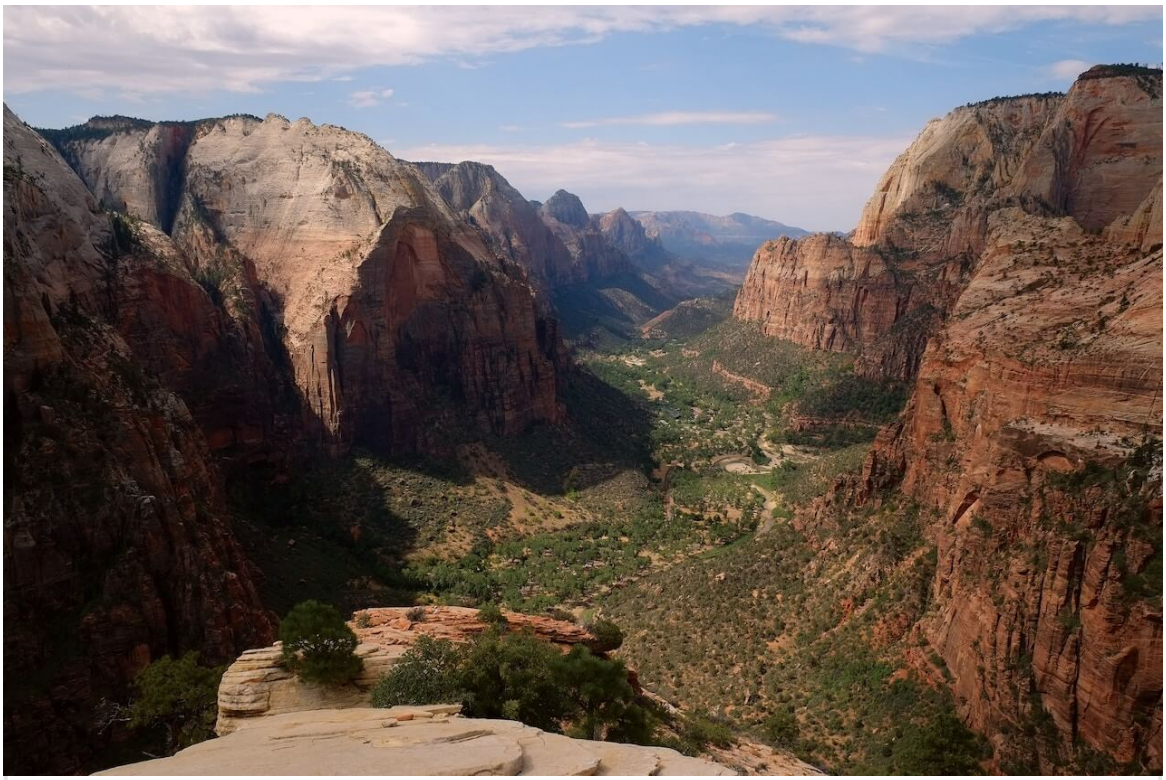
The `np.unique` method can be used to get the unique items. If you want the repetition counts of each item, set the return_counts parameter to True.

```
# Create random integers of size 10 between [0,10)
np.random.seed(100)
arr_rand = np.random.randint(0, 10, size=10)
print(arr_rand)

#> [8 8 3 7 7 0 4 2 5 2]
```

```
# Get the unique items and their counts
uniqs, counts = np.unique(arr_rand, return_counts=True)
print("Unique items : ", uniqs)
print("Counts       : ", counts)

#> Unique items :  [0 2 3 4 5 7 8]
#> Counts       :  [1 2 1 1 1 2 2]
```

Numpy Tutorial Part 2 – Vital Functions for Data Analysis

Numpy is the core package for data analysis and scientific computing in python. This is part 2 of a mega numpy tutorial. In this part, I go into the details of the advanced features of numpy that are essential for data analysis and manipulations.



Numpy Tutorial Part 2: Vital Functions for Data Analysis. Photo by Ana Philipa Neves.

Contents

Introduction

In part 1 of the numpy tutorial [https://www.machinelearningplus.com/numpy-tutorial-part1-array-python-examples/] we got introduced to numpy and why its so important to know numpy if you are to work with datasets in python. In particular, we discussed how to create arrays, explore it, indexing, reshaping, flattening, generating random numbers and many other functions.

In part 2 [this tutorial], I continue from where we left and take it up a notch by dealing with slightly more advanced but essential topics for data analysis.

I will assume that you have some familiarity with python, know basic math and have already read the part 1 of the numpy tutorial.

The best way to approach this post is to read the whole article fairly quick in one go and then come back to the beginning and try out the examples in a jupyter notebook.

Let's begin.

1. How to get index locations that satisfy a given condition using np.where?

Previously you saw how to extract items from an array that satisfy a given condition. Boolean indexing, remember?

But sometimes we want to know the index positions of the items (that satisfy a condition) and do whatever you want with it.

`np.where` locates the positions in the array where a given condition holds true.

```
# Create an array
import numpy as np
arr_rand = np.array([8, 8, 3, 7, 7, 0, 4, 2, 5, 2])
print("Array: ", arr_rand)

# Positions where value > 5
index_gt5 = np.where(arr_rand > 5)
print("Positions where value > 5: ", index_gt5)
```

```
#> Array:  [8 8 3 7 7 0 4 2 5 2]
#> Positions where value > 5:  (array([0, 1, 3, 4]),)
```

Once you have the positions, you can extract them using the array's `take` method.

```
# Take items at given index
arr_rand.take(index_gt5)
```

```
#> array([[8, 8, 7, 7]])
```

Thankfully, `np.where` also accepts 2 more optional arguments x and y. Whenever condition is true, 'x' is yielded else 'y'.

Below, I try to create an array that will have the string 'gt5' whenever the condition is true, else, it will have 'lt5'.

```
# If value > 5, then yield 'gt5' else 'le5'
np.where(arr_rand > 5, 'gt5', 'le5')
```

```
#> array(['gt5', 'gt5', 'le5', 'gt5', 'gt5', 'le5', 'le5', 'le5', 'le5', 'le5'],
      dtype='<U3')
```

Let's find the location of the maximum and minimum valjues as well.

```
# Location of the max
print('Position of max value: ', np.argmax(arr_rand))

# Location of the min
print('Position of min value: ', np.argmin(arr_rand))
```

```
#> Position of max value:  0
#> Position of min value:  5
```

Good.

2. How to import and export data as a csv file?

A standard way to import datasets is to use the `np.genfromtxt` function. It can import datasets from web URLs, handle missing values, multiple delimiters, handle irregular number of columns etc.

A less versatile version is the `np.loadtxt` which assumes the dataset has no missing values.

As an example, let's try to read a .csv file [http://'https://raw.githubusercontent.com/selva86/datasets/master/Auto.csv'] from the below URL. Since all elements in a numpy array should be of the same data type, the last column which is a text will be imported as a 'nan' by default.

By setting the `filling_values` argument you can replace the missing values with something else.

```
# Turn off scientific notation
np.set_printoptions(suppress=True)

# Import data from csv file url
path = 'https://raw.githubusercontent.com/selva86/datasets/master/Auto.csv'
data = np.genfromtxt(path, delimiter=',', skip_header=1, filling_values=-999, dtype='f
loat')
data[:3]  # see first 3 rows
```

```
#> array([[   18. ,     8. ,   307. ,   130. ,  3504. ,    12. ,    70. ,
#>             1. ,  -999. ],
#>        [   15. ,     8. ,   350. ,   165. ,  3693. ,    11.5,    70. ,
#>             1. ,  -999. ],
#>        [   18. ,     8. ,   318. ,   150. ,  3436. ,    11. ,    70. ,
#>             1. ,  -999. ]])
```

That was neat. But did you notice all the values in last column has the same value '-999'?

That happened because, I had mentioned the. `dtype='float'`. The last column in the file contained text values and since all the values in a numpy array has to be of the same `dtype`, `np.genfromtxt` didn't know how to convert it to a float.

2.1 How to handle datasets that has both numbers and text columns?

In case, you MUST have the text column as it is without replacing it with a placeholder, you can either set the dtype as 'object' or as None.

```
# data2 = np.genfromtxt(path, delimiter=',', skip_header=1, dtype='object')
data2 = np.genfromtxt(path, delimiter=',', skip_header=1, dtype=None)
data2[:3]  # see first 3 rows
```

```
#> array([( 18., 8,  307., 130, 3504,  12. , 70, 1, b'"chevrolet chevelle malibu"'),
#>        ( 15., 8,  350., 165, 3693,  11.5, 70, 1, b'"buick skylark 320"'),
#>        ( 18., 8,  318., 150, 3436,  11. , 70, 1, b'"plymouth satellite"')],
#>       dtype=[('f0', '<f8'), ('f1', '<i8'), ('f2', '<f8'), ('f3', '<i8'), ('f4', '<i
8'), ('f5', '<f8'), ('f6', '<i8'), ('f7', '<i8'), ('f8', 'S38')])
```

Excellent!

Finally, 'np.savetxt' lets you export the array as a csv file.

```
# Save the array as a csv file
np.savetxt("out.csv", data, delimiter=",")
```

3. How to save and load numpy objects?

At some point, we will want to save large transformed numpy arrays to disk and load it back to console directly without having the re-run the data transformations code.

Numpy provides the .npy and the .npz file types for this purpose.

If you want to store a single `ndarray` object, store it as a `.npy` file using `np.save`. This can be loaded back using the `np.load`.

If you want to store more than 1 `ndarray` object in a single file, then save it as a `.npz` file using `np.savez`.

```
# Save single numpy array object as .npy file
np.save('myarray.npy', arr2d)

# Save multile numy arrays as a .npz file
np.savez('array.npz', arr2d_f, arr2d_b)
```

Load back the .npy file.

```
# Load a .npy file
a = np.load('myarray.npy')
print(a)
```

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

Load back the .npz file.

```
# Load a .npz file
b = np.load('array.npz')
print(b.files)
b['arr_0']
```

```
#> ['arr_0', 'arr_1']

#> array([[ 0.,  1.,  2.],
#>        [ 3.,  4.,  5.],
#>        [ 6.,  7.,  8.]])
```

4. How to concatenate two numpy arrays columnwise and row wise

There are 3 different ways of concatenating two or more numpy arrays.

- Method 1: `np.concatenate` by changing the axis parameter to 0 and 1
- Method 2: `np.vstack` and `np.hstack`
- Method 3: `np.r_` and `np.c_`

All three methods provide the same output.

One key difference to notice is unlike the other 2 methods, both `np.r_` and `np.c_` use square brackets to stack arrays. But first, let me create the arrays to be concatenated.

```
a = np.zeros([4, 4])
b = np.ones([4, 4])
print(a)
print(b)
```

```
#> [[ 0.  0.  0.  0.]
#>  [ 0.  0.  0.  0.]
#>  [ 0.  0.  0.  0.]
#>  [ 0.  0.  0.  0.]]

#> [[ 1.  1.  1.  1.]
#>  [ 1.  1.  1.  1.]
#>  [ 1.  1.  1.  1.]
#>  [ 1.  1.  1.  1.]]
```

Let's stack the arrays vertically.

```
# Vertical Stack Equivalents (Row wise)
np.concatenate([a, b], axis=0)
np.vstack([a,b])
np.r_[a,b]
```

```
#> array([[ 0.,  0.,  0.,  0.],
#>        [ 0.,  0.,  0.,  0.],
#>        [ 0.,  0.,  0.,  0.],
#>        [ 0.,  0.,  0.,  0.],
#>        [ 1.,  1.,  1.,  1.],
#>        [ 1.,  1.,  1.,  1.],
#>        [ 1.,  1.,  1.,  1.],
#>        [ 1.,  1.,  1.,  1.]])
```

That was what we wanted. Let's do it horizontally (columns wise) as well.

```
# Horizontal Stack Equivalents (Coliumn wise)
np.concatenate([a, b], axis=1)
np.hstack([a,b])
np.c_[a,b]
```

```
#> array([[ 0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.],
#>        [ 0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.],
#>        [ 0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.],
#>        [ 0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.]])
```

Besides, you can use `np.r_` to create more complex number sequences in 1d arrays.

```
np.r_[[1,2,3], 0, 0, [4,5,6]]
```

```
#> array([1, 2, 3, 0, 0, 4, 5, 6])
```

5. How to sort a numpy array based on one or more columns?

Let's try and sort a 2d array based on the first column.

```
arr = np.random.randint(1,6, size=[8, 4])
arr
```

```
#> array([[3, 3, 2, 1],
#>        [1, 5, 4, 5],
#>        [3, 1, 4, 2],
#>        [3, 4, 5, 5],
#>        [2, 4, 5, 5],
#>        [4, 4, 4, 2],
#>        [2, 4, 1, 3],
#>        [2, 2, 4, 3]])
```

We have a random array of 8 rows and 4 columns.

If you use the `np.sort` function with `axis=0`, all the columns will be sorted in ascending order independent of eachother, effectively compromising the integrity of the row items. In simple terms, the values in each row gets corrupted with values from other rows.

```
# Sort each columns of arr
np.sort(arr, axis=0)
```

```
#> array([[1, 1, 1, 1],
#>        [2, 2, 2, 2],
#>        [2, 3, 4, 2],
#>        [2, 4, 4, 3],
#>        [3, 4, 4, 3],
#>        [3, 4, 4, 5],
#>        [3, 4, 5, 5],
#>        [4, 5, 5, 5]])
```

Since I don't want the content of rows to be disturbed, I resort to an indirect method using `np.argsort`.

5.1 How to sort a numpy array based on 1 column using argsort?

Let's first understand what `np.argsort` does.

`np.argsort` returns the index positions of that would make a given 1d array sorted.

```
# Get the index positions that would sort the array
x = np.array([1, 10, 5, 2, 8, 9])
sort_index = np.argsort(x)
print(sort_index)
```

```
#> [0 3 2 4 5 1]
```

How to interpret this?

In array ‘x’, the 0th item is the smallest, 3rd item is the second smallest and so on.

```
x[sort_index]
```

```
#> array([ 1,  2,  5,  8,  9, 10])
```

Now, in order to sort the original `arr`, I am going to do an argsort on the 1st column and use the resulting index positions to sort `arr`. See the code.

```
# Argsort the first column
sorted_index_1stcol = arr[:, 0].argsort()

# Sort 'arr' by first column without disturbing the integrity of rows
arr[sorted_index_1stcol]
```

```
#> array([[1, 5, 4, 5],
#>        [2, 4, 5, 5],
#>        [2, 4, 1, 3],
#>        [2, 2, 4, 3],
#>        [3, 3, 2, 1],
#>        [3, 1, 4, 2],
#>        [3, 4, 5, 5],
#>        [4, 4, 4, 2]])
```

To sort it in decreasing order, simply reverse the argsorted index.

```
# Descending sort
arr[sorted_index_1stcol[::-1]]
```

```
#> array([[4, 4, 4, 2],
#>        [3, 4, 5, 5],
#>        [3, 1, 4, 2],
#>        [3, 3, 2, 1],
#>        [2, 2, 4, 3],
#>        [2, 4, 1, 3],
#>        [2, 4, 5, 5],
#>        [1, 5, 4, 5]])
```

5.2 How to sort a numpy array based on 2 or more columns?

You can do this using `np.lexsort` by passing a tuple of columns based on which the array should be sorted.

Just remember to place the column to be sorted first at the rightmost side inside the tuple.

```
# Sort by column 0, then by column 1
lexsorted_index = np.lexsort((arr[:, 1], arr[:, 0]))
arr[lexsorted_index]
```

```
#> array([[1, 5, 4, 5],
#>        [2, 2, 4, 3],
#>        [2, 4, 5, 5],
#>        [2, 4, 1, 3],
#>        [3, 1, 4, 2],
#>        [3, 3, 2, 1],
#>        [3, 4, 5, 5],
#>        [4, 4, 4, 2]])
```

6. Working with dates

Numpy implements dates through the `np.datetime64` object which supports a precision till nanoseconds. You can create one using a standard YYYY-MM-DD formatted date strings.

```
# Create a datetime64 object
date64 = np.datetime64('2018-02-04 23:10:10')
date64
```

```
#> numpy.datetime64('2018-02-04T23:10:10')
```

Ofcourse you can pass hours, minutes, seconds till nanoseconds as well.

Let's remove the time component from `date64`.

```
# Drop the time part from the datetime64 object
dt64 = np.datetime64(date64, 'D')
dt64
```

```
#> numpy.datetime64('2018-02-04')
```

By default, if you add a number increases the number of days. But if you need to increase any other time unit like months, hours, seconds etc, then the `timedelta` object is much convenient.

```
# Create the timedeltas (individual units of time)
tenminutes = np.timedelta64(10, 'm')  # 10 minutes
tenseconds = np.timedelta64(10, 's')  # 10 seconds
tennanoseconds = np.timedelta64(10, 'ns')  # 10 nanoseconds

print('Add 10 days: ', dt64 + 10)
print('Add 10 minutes: ', dt64 + tenminutes)
print('Add 10 seconds: ', dt64 + tenseconds)
print('Add 10 nanoseconds: ', dt64 + tennanoseconds)
```

```
#> Add 10 days:  2018-02-14
#> Add 10 minutes:  2018-02-04T00:10
#> Add 10 seconds:  2018-02-04T00:00:10
#> Add 10 nanoseconds:  2018-02-04T00:00:00.000000010
```

Let me convert the `dt64` back to a string.

```
# Convert np.datetime64 back to a string
np.datetime_as_string(dt64)
```

```
#> '2018-02-04'
```

When working with dates, you would often need to filter out the business days from the data. You can know if a given date is a business day or not using the `np.is_busday()`.

```
print('Date: ', dt64)
print("Is it a business day?: ", np.is_busday(dt64))
print("Add 2 business days, rolling forward to nearest biz day: ", np.busday_offset(dt64, 2, roll='forward'))
print("Add 2 business days, rolling backward to nearest biz day: ", np.busday_offset(dt64, 2, roll='backward'))
```

```
#> Date:  2018-02-04
#> Is it a business day?:  False
#> Add 2 business days, rolling forward to nearest biz day:  2018-02-07
#> Add 2 business days, rolling backward to nearest biz day:  2018-02-06
```

6.1 How to create a sequence of dates?

It can simply be done using the `np.arange` itself.

```
# Create date sequence
dates = np.arange(np.datetime64('2018-02-01'), np.datetime64('2018-02-10'))
print(dates)

# Check if its a business day
np.is_busday(dates)
```

```
#> ['2018-02-01' '2018-02-02' '2018-02-03' '2018-02-04' '2018-02-05'
#>  '2018-02-06' '2018-02-07' '2018-02-08' '2018-02-09']

array([ True,  True, False, False,  True,  True,  True,  True,  True], dtype=bool)
```

6.2 How to convert numpy.datetime64 to datetime.datetime object?

```
# Convert np.datetime64 to datetime.datetime
import datetime
dt = dt64.tolist()
dt
```

```
#> datetime.date(2018, 2, 4)
```

Once you convert it to a `datetime.date` object, you have a lot more facilities to extract the day of month, month of year etc.

```
print('Year: ', dt.year)
print('Day of month: ', dt.day)
print('Month of year: ', dt.month)
print('Day of Week: ', dt.weekday())  # Sunday
```

```
#> Year:  2018
#> Day of month:  4
#> Month of year:  2
#> Day of Week:  6
```

7. Advanced numpy functions

7.1 vectorize – Make a scalar function work on vectors

With the help of `vectorize()` you can make a function that is meant to work on individual numbers, to work on arrays.

Let's see a simplified example.

The function `foo` (see code below) accepts a number and squares it if it is 'odd' else it divides it by 2.

When you apply this function on a scalar (individual numbers) it works perfectly, but fails when applied on an array.

With numpy's `vectorize()`, you can magically make it work on arrays as well.

```
# Define a scalar function
def foo(x):
    if x % 2 == 1:
        return x**2
    else:
        return x/2

# On a scalar
print('x = 10 returns ', foo(10))
print('x = 11 returns ', foo(11))

# On a vector, doesn't work
# print('x = [10, 11, 12] returns ', foo([10, 11, 12]))  # Error
```

```
#> x = 10 returns  5.0
#> x = 11 returns  121
```

Let's vectorize `foo()` so it will work on arrays.

```
# Vectorize foo(). Make it work on vectors.
foo_v = np.vectorize(foo, otypes=[float])

print('x = [10, 11, 12] returns ', foo_v([10, 11, 12]))
print('x = [[10, 11, 12], [1, 2, 3]] returns ', foo_v([[10, 11, 12], [1, 2, 3]]))
```

```
#> x = [10, 11, 12] returns  [   5.  121.    6.]
#> x = [[10, 11, 12], [1, 2, 3]] returns  [[   5.  121.    6.]
#>  [   1.    1.    9.]]
```

This can be very handy whenever you want to make a scalar function work on arrays.

`vectorize` also accepts an optional `otypes` parameter where you provide what the datatype of the output should be. It makes the vectorized function run faster.

7.2 apply_along_axis – Apply a function column wise or row wise

Let me first create a 2D array to show this.

```
# Create a 4x10 random array
np.random.seed(100)
arr_x = np.random.randint(1,10,size=[4,10])
arr_x
```

```
#> array([[9, 9, 4, 8, 8, 1, 5, 3, 6, 3],
#>        [3, 3, 2, 1, 9, 5, 1, 7, 3, 5],
#>        [2, 6, 4, 5, 5, 4, 8, 2, 2, 8],
#>        [8, 1, 3, 4, 3, 6, 9, 2, 1, 8]])
```

Let's understand this by solving the following question:

How to find the difference of the maximum and the minimum value in each row?

Well, the normal approach would be to write a for-loop that iterates along each row and then compute the max-min in each iteration.

That sounds alright but it can get cumbersome if you want to do the same column wise or want to implement a more complex computation. Besides, it can consume more keystrokes.

You can do this elegantly using the `numpy.apply_along_axis`.

It takes as arguments:

1. Function that works on a 1D vector [`fund1d`]
2. Axis along which to apply `func1d`. For a 2D array, 1 is row wise and 0 is column wise.
3. Array on which `func1d` should be applied.

Let's implement this.

```
# Define func1d
def max_minus_min(x):
    return np.max(x) - np.min(x)

# Apply along the rows
print('Row wise: ', np.apply_along_axis(max_minus_min, 1, arr=arr_x))

# Apply along the columns
print('Column wise: ', np.apply_along_axis(max_minus_min, 0, arr=arr_x))
```

```
#> Row wise:  [8 8 6 8]
#> Column wise:  [7 8 2 7 6 5 8 5 5 5]
```

7.3 searchsorted – Find the location to insert so the array will remain sorted

what does `numpy.searchsorted` do?

It gives the index position at which a number should be inserted in order to keep the array sorted.

```
# example of searchsorted
x = np.arange(10)
print('Where should 5 be inserted?: ', np.searchsorted(x, 5))
print('Where should 5 be inserted (right)?: ', np.searchsorted(x, 5, side='right'))
```

```
#> Where should 5 be inserted?:  5
#> Where should 5 be inserted (right)?:  6
```

With the smart hack [https://twitter.com/RadimRehurek/status/928671225861296128] by Radim, you can use searchsorted to do sampling elements with probabilities. It's much faster than `np.choice`.

```
# Randomly choose an item from a list based on a predefined probability
lst = range(10000)  # the list
probs = np.random.random(10000); probs /= probs.sum()  # probabilities

%timeit lst[np.searchsorted(probs.cumsum(), np.random.random())]
%timeit np.random.choice(lst, p=probs)
```

```
#> 36.6 µs ± 3.93 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
#> 1.02 ms ± 7.16 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

7.4 How to add a new axis to a numpy array?

Sometimes you might want to convert a 1D array into a 2D array (like a spreadsheet) without adding any additional data.

You might need this in order a 1D array as a single column in a csv file, or you might want to concatenate it with another array of similar shape.

Whatever the reason be, you can do this by inserting a new axis using the `np.newaxis`.

Actually, using this you can raise an array of a lower dimension to a higher dimension.

```
# Create a 1D array
x = np.arange(5)
print('Original array: ', x)

# Introduce a new column axis
x_col = x[:, np.newaxis]
print('x_col shape: ', x_col.shape)
print(x_col)

# Introduce a new row axis
x_row = x[np.newaxis, :]
print('x_row shape: ', x_row.shape)
print(x_row)
```

```
#> Original array:  [0 1 2 3 4]
#> x_col shape:  (5, 1)
#> [[0]
#>  [1]
#>  [2]
#>  [3]
#>  [4]]
#> x_row shape:  (1, 5)
#> [[0 1 2 3 4]]
```

7.5 More Useful Functions

Digitize

Use `np.digitize` to return the index position of the bin each element belongs to.

```
# Create the array and bins
x = np.arange(10)
bins = np.array([0, 3, 6, 9])

# Get bin allotments
np.digitize(x, bins)
```

```
#> array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4])
```

Clip

Use np.clip to cap the numbers within a given cutoff range. All number lesser than the lower limit will be replaced by the lower limit. Same applies to the upper limit also.

```
# Cap all elements of x to lie between 3 and 8
np.clip(x, 3, 8)
```

```
#> array([3, 3, 3, 3, 4, 5, 6, 7, 8, 8])
```

Histogram and Bincount

Both `histogram()` and `bincount()` gives the frequency of occurences. But with certain differences.

While `histogram()` gives the frequency counts of the bins, `bincount()` gives the frequency count of all the elements in the range of the array between the min and max values. Including the values that did not occur.

```
# Bincount example
x = np.array([1,1,2,2,2,4,4,5,6,6,6]) # doesn't need to be sorted
np.bincount(x) # 0 occurs 0 times, 1 occurs 2 times, 2 occurs thrice, 3 occurs 0 time
s, ...

# Histogram example
counts, bins = np.histogram(x, [0, 2, 4, 6, 8])
print('Counts: ', counts)
print('Bins: ', bins)
```

```
#> Counts:  [2 3 3 3]
#> Bins:  [0 2 4 6 8]
```

8. What is missing in numpy?

So far we have covered a good number of techniques to do data manipulations with numpy. But there are a considerable number of things you can't do with numpy directly. At least to my limited knowledge. Let me list a few:

1. No direct function to merge two 2D arrays based on a common column.
2. Create pivot tables directly
3. No direct way of doing 2D cross tabulations.
4. No direct method to compute statistics (like mean) grouped by unique values in an array.
5. And more..

Well, the reason I am telling you this is these shortcomings are nicely handled by the spectacular pandas library which I will talk about in the upcoming pandas tutorial.

Meanwhile, you might want to test your skills with the numpy practice exercises (https://www.machinelearningplus.com/101-numpy-exercises-python/).

You may also like:

P-Value Intuition with Clear Explanation

2_Bubble_Plot_Matplotlib

LDA - How to grid search best topic models? (with examples in python)

Python debugging with pdb