Python: NumPy Course Notes

Klein
carlj.klein@gmail.com

1 Learning Objectives

Concepts

- Creating and Saving NumPy ndarrays
- Using Built-in Functions to Create ndarrays
- Accessing, Deleting, and Inserting Elements Into ndarrays
- Slicing ndarrays
- Boolean Indexing, Set Operations, and Sorting
- Exercise: Manipulating ndarrays
- Arithmetic Operations and Broadcasting
- Exercise: Creating ndarrays with Broadcasting

Commands

- np.array
- .shape
- .size
- np.save and np.load
- np.zeros
- np.ones
- np.eye
- np.diag
- np.arange
- np.linspace
- np.reshape

- np.random
  - np.random.random
  - np.random.randint
  - np.random.normal
  - np.random.permutation
- np.delete
- np.append
- np.insert
- np.vstack
- np.hstack
- np.unique
- np.copy
- np.intersect1d
- np.setdiff1d
- np.union1d
- np.sort
- Mathematical Functions
  - np.add
  - np.subtract
  - np.multiply
  - np.divide
  - np.sqrt
  - np.exp
  - np.power
- Statistical Functions
  - mean
  - std
  - median
  - max
  - min

NumPy is useful in dealing with arrays of data, which can be thought of in a mathematical sense as vectors and matrices. NumPy is a powerful tool in itself, however an even more useful Python data tool, the Pandas library, is built on top of NumPy. A basic understanding of NumPy is necessary, and we'll cover some of the essential functions and uses in these notes.

2 Creating and Saving NumPy ndarrays

Not needing too much of an explanation, here's a beginning look at creating basic NumPy arrays.

Covered in this section:

- np.array: turn a list, or list of lists, into a NumPy array object
- shape: return the dimensions of an array object
- size: return the total number of elements in an array object
- save & load: arrays can be saved for future use

```
x = np.array([1 ,2, 3, 4, 5])
print(x)
# [1 2 3 4 5]

x.shape
# (5,)

Y = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(Y)
"""
[[ 1  2  3]
[ 4  5  6]
[ 7  8  9]
[10 11 12]]
"""

Y.shape
# (4, 3)
Y.size
# 12

# Notes Section:
# dtype changes with mixed data, and will uptype (int + floats ->
                                    all floats)
# numpy arrays can be saved for later use

# save
np.save('my_array', x)
# load
y = np.load('my_array.npy')
```

3 Using Built-in Functions to Create ndarrays

There are several function in the NumPy library which build generalized arrays, or arrays populated by data generated following rules. For example, build an array with dimensions nxm that consist of all 0s, or 1s along the diagonal, etc.

Here are some of the essential generating functions:

- zeros: create an array populated with all 0s
- ones: create an array populated with all 1s
- full: create an array populated with a specified input
- eye: creates an array populated with the identiy matrix (s1 on the diagonals, 0s for the rest)
- diag: takes a list, and places the list elements on the diagnoal of a matrix
- arange: will return a 1d array with increasing values based upon specified start, stop and spacing where the endpoint is non-inclusive: [start, stop)
- linspace: will return a 1d array with increasing values based upon start, stop and number of values desired where the endpoint is inclusive: [start, stop]
- reshape: will return an array with the elements reshaped into desired dimensions (given that the two arrays have a compatible number of elements)
- np.random.random: will return an array of random numbers filling specified dimensions
- np.random.randint: will return an array of random integers based upon a range, and filling specified dimensions
- np.random.normal: will return an array of normally distributed values based upon a range, and filling specified dimensions (note that this concept transfers to other common distributions)

```
# numpy zeros function
x = np.zeros((3, 4))
# can change dtype: np.zeros((3, 4), dtype = int)

# Below is what x, the object, look like:
"""
array([[0., 0., 0., 0.],
[0., 0., 0., 0.],
[0., 0., 0., 0.]])
"""
# For the rest of the examples, we'll be showing the print(x)
                                version:
"""
[[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]
"""
```

```
x = np.ones((4, 5))
"""
[[1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1.]]
"""

x = np.full((4, 3), 5)
"""
[[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]]
"""

x = np.eye(5)
"""
[[1. 0. 0. 0. 0.]
[0. 1. 0. 0. 0.]
[0. 0. 1. 0. 0.]
[0. 0. 0. 1. 0.]
[0. 0. 0. 0. 1.]]
"""

x = np.diag([1, 2, 3, 4, 5])
"""
[[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
[0 0 0 4 0]
[0 0 0 0 5]]
"""

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

x = np.arange(4, 10)
"""
# [4 5 6 7 8 9]
"""

x = np.arange(1, 14, 3)
"""
# [ 1  4  7 10 13]
"""

# numpy linspace function (default = 50)
x = np.linspace(0, 25, 10)
"""
[ 0.          2.77777778  5.55555556  8.33333333 11.11111111 13.
                                    88888889
16.66666667 19.44444444 22.22222222 25.        ]
"""
```

```
x = np.linspace(0, 25, 10, endpoint=False)
"""
[ 0.   2.5  5.   7.5 10.  12.5 15.  17.5 20.  22.5]
"""

# numpy array methods
x = np.arange(20).reshape((4, 5))
"""
[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]]
"""

# numpy random functions
x = np.random.random((3, 3))
"""
[[0.82591877 0.69755492 0.27402617]
[0.13409014 0.66530258 0.10349078]
[0.22057372 0.17607417 0.98691068]]
"""

x = np.random.randint(4, 15, (3, 2))
"""
[[ 9  4]
[14  8]
[ 4  5]]
"""

# the 1000x1000 example is not illustrated, for formatting
                                    reasons
x = np.random.normal(0, 0.1, size = (1000, 1000))
```

4 Accessing, Deleting, and Inserting Elements Into ndarrays

NumPy arrays are mutable objects, meaning they can be altered following creation. We'll be going over several ways to access and change existing arrays. Some of the essential functions covered are:

- delete: specify the array and the indices for elements to be deleted
- append: add elements to the end of an array
- insert: add elements at a specified position of an array
- vstack: stack two NumPy arrays vertically
- hstack: stack two NumPy arrays horizontally

```
# Can access and change elements based on indices
x = np.arange(1, 10).reshape(3, 3)
"""
[[1 2 3]
[4 5 6]
[7 8 9]]
"""
# Access element
x[0, 0] = 1
# 1

# Change element
x[0, 0] = 99
"""
[[99  2  3]
[ 4  5  6]
[ 7  8  9]]
"""

# deleting elements
x = np.arange(1, 6)
# [1 2 3 4 5]
# delete first and last elements
x = np.delete(x, [0, -1])
# [2 3 4]

x = np.arange(1, 10).reshape(3, 3)
# for rank 2+ arrays, axis refers to dimension (row = 0, column =
                                    1)
W = np.delete(x, 0, axis = 0)
"""
W is x with first row deleted
[[4 5 6]
[7 8 9]]
"""
W = np.delete(x, [0, 2], axis = 1)
"""
W is x with first and last columns deleted
[[2]
[5]
[8]]
"""

# adding (appending) elements
x = np.arange(1, 6)
x = np.append(x, [6, 7])
# [1 2 3 4 5 6 7]
x = np.arange(1, 10).reshape(3, 3)
W = np.append(x, [[10 11 12]], axis = 0) # append row
Y = np.append(x, [[10], [11], [12]], axis = 1) # append column

# inserting elements
x = np.arange(1, 6)
# insert elements in between elements in rank 1 array
x = np.insert(x, 2, [3 ,4])
print(x)
# [1 2 3 4 3 4 5]
```

```
# insert row between rows in rank 2 array
x = np.arange(1, 6)
Y = np.arange(1, 10).reshape(3, 3)
W = np.insert(Y, 1, [4, 5, 6], axis = 0)
print(W)
"""
[[1 2 3]
[4 5 6]
[4 5 6]
[7 8 9]]
"""

# insert column between columns in rank 2 array
W = np.insert(Y, 1, 5, axis = 1)
print(W)
"""
[[1 5 2 3]
[4 5 5 6]
[7 5 8 9]]
"""

# stack arrays using vertical or horizontal stacking (pay
                                    attention to dimension)
x = np.array([1, 2])
Y = np.array([[3, 4], [5,6]])
z = np.vstack((x, Y))
print(z)
"""
[[1 2]
[3 4]
[5 6]]
"""

z = np.hstack((Y, x.reshape(2, 1)))
print(z)
"""
[[3 4 1]
[5 6 2]]
"""
```

5 Slicing ndarrays

Covered briefly in the previous section, we're going to do a deeper dive into the slicing capabilities of NumPy. Aside from syntax, some essential functions covered in this section are:

- copy: this creates an entirely new array, where changes won't be mapped back to the original
- diag: another use of the diagonal function, this is used to return an array featuring the diagonal of a rank 2 array
- unique: this will find all of the unique values within an array

```
# Here are the basic ways to slice a rank 1 array:
# 1. ndarray[start:end]
# 2. ndarray[start:]
# 3. ndarray[:end]
# Also, recall that ndarray[-1] will return the last element in
                                an array

# examples of slicing in rank 2 array
X = np.arange(1, 21).reshape(4, 5)
"""
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]
"""

# the rank 2 slice order is: rows, columns (starts are inclusive,
                                ends are exclusive)
z = X[1:4, 2:5]
"""
[[ 8  9 10]
[13 14 15]
[18 19 20]]
"""

# by leaving either the row or column input as ":" it will select
                                entire row or column
# this will return the 3rd column
z = X[:, 2]
# [ 3  8 13 18]
# however, that returns a rank 1 array (row format). Let's return
                                the correct dimensions:
z = X[:, 2:3]
"""
[[ 3]
[ 8]
[13]
[18]]
"""

# slicing creates a "view" of the array
# changing z will change X (keep the copy command in mind here)
z = X[1:, 2:]
z[2, 2] = 0
"""
X =
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19  0]]

Even though the elements were changed in the z array, there was
                                still a connection to the X array
                                .
"""

# create a brand new array (so to not change the original array)
```

```
z = X[1:, 2:].copy()
z[2, 2] = 0
"""
X =
[[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]]

By using the copy function the connection to the original array
                                    was broken.
"""

# use variable in slicing
indices = np.array([1, 3])
y = X[indices, :]
"""
[[ 6  7  8  9 10]
[16 17 18 19 20]]
"""
z = X[:, indices]
"""
[[ 2  4]
[ 7  9]
[12 14]
[17 19]]
"""

# NumPy built in slicing methods
# ex: diagonals
z = np.diag(X)
# [ 1  7 13 19]
# specify k to shift the diagonal indexing
z = np.diag(X, k = 1)
# [ 2  8 14 20]

# finding uniques
X = np.array([[1, 2, 3], [4, 5, 6], [1, 5, 6]])
# [1 2 3 4 5 6]
```

6 Boolean Indexing, Set Operations, and Sorting

Now that we have a solid understanding of elements and slicing in rank 1 and rank 2 arrays, we can move into conditionals and comparing arrays with each other. Some essentials covered in this section:

- intersect1d: this will find the intersection of two arrays and return the sorted, unique values
- setdiff1d: this will find the differences between two arrays and return the sorted, unique values

- union1d: this will find the union between two arrays and return the sorted, unique values
- sort: this will sort the values of an array

```
X = np.arange(25).reshape(5, 5)
"""
[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]
[20 21 22 23 24]]
"""

# first look at conditional arguments
X[X > 10]
X[(X > 10) & (X < 17)]
"""
[11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[11 12 13 14 15 16]
"""

# basic application of conditional arguments (other than slicing)
X[(X > 10) & (X < 17)] = -1
"""
[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 -1 -1 -1 -1]
[-1 -1 17 18 19]
[20 21 22 23 24]]
"""

x = np.array([1, 2, 3, 4])
y = np.array([3, 4, 5, 6])
np.intersect1d(x, y)
np.setdiff1d(x, y)
np.union1d(x, y)
"""
Recall these functions return sorted, unique arrays

Intersection
[3 4]

Difference
[1 2]

Union
[1 2 3 4 5 6]
"""
x = np.random.randint(1, 11, size = (10,))
np.sort(x)
"""
np.sort(x) =
[2 2 2 2 3 5 5 5 7 9]

x =
[2 5 2 5 5 3 2 9 7 2]
```

```
"""

np.sort(np.unique(x))
"""
[2 3 5 7 9]
"""

# can also sort using x.sort()
x.sort()
"""
x =
[2 2 2 2 3 5 5 5 7 9]

Note the difference between np.sort(x) and x.sort():
- np.sort(x) creates an instance where x is sorted
- x.sort() changes the array to be sorted
"""

x = np.random.randint(1, 11, size = (5,5))
"""
[[ 9  2 10  6  6]
[10  7  6 10  8]
[ 3  1  1  4  2]
[ 9  6  5  7  3]
[10  7  5  7  7]]
"""

# can sort 2d arrays using the axis command
# sort rows with axis = 0, columns with axis = 1
np.sort(x, axis = 0)
np.sort(x, axis = 1)
"""
axis = 0 (row sort)
[[ 3  1  1  4  2]
[ 9  2  5  6  3]
[ 9  6  5  7  6]
[10  7  6  7  7]
[10  7 10 10  8]]

axis = 1 (column sort)
[[ 2  6  6  9 10]
[ 6  7  8 10 10]
[ 1  1  2  3  4]
[ 3  5  6  7  9]
[ 5  7  7  7 10]]
"""
```

It's important to note that we should be careful and deliberate when using the sort function. For instance, say we have a rank 2 array that is comprised of data entries where either each row or each column are data points referring to a specific event. If not done properly, sorting could mix the data from the events.

7 Exercise: Manipulating ndarrays

This is a quick exercise to test the combination of some of the NumPy functions we've seen so far.

The task is to create a 5x5 ndarray with consecutive integers from 1 to 25 (inclusive). Then, use Boolean indexing to pick out only the odd numbers in the array.

```
# Create a 5 x 5 ndarray with consecutive integers from 1 to 25 (
                                inclusive).
X = np.arange(1, 26).reshape(5, 5)

# Use Boolean indexing to pick out only the odd numbers in the
                                array
Y = X[X % 2 != 0]
```

Although a short exercise in code length, this illustrates a power conditional argument when finding even and odd values, known in mathematics as the "modulus" or "mod" for short. Essentially, the command above divides the numbers by 2, and checks to see if there is any remainder. If there is no remainder, the number is even and is accepted.

8 Arithmetic Operations and Broadcasting

Another feature of NumPy that makes it so powerful, is its built-in ability to perform mathematical operations both element-wise and matrix-wise. The concepts of the built-in mathematical functions range, but here are some of the essentials:

- add
- subtract
- multiply
- divide
- sqrt
- exp
- power
- Other Mathematical Functions
- Statistical Functions
- Broadcasting

Not exactly a function in itself, there is also the idea of broadcasting that will be covered in this section. Broadcasting refers to how NumPy handles arrays of different dimensions while performing arithmetic operations. NumPy generally broadcasts the smaller array across the larger array in order ot have compatible shapes. When performing arithmetic operations in the case of arrays with the same shape, the operations are done so on corresponding elements between the arrays. In the case of different shapes, one of the dimensions should be equivalent between the two arrays, and then the smaller array's values are extended either row-wise or column-wise to allow for the operation. An example will be provided in the code.

```
# NumPy allows element-wise and matrix-wise operations
x = np.array([1, 2, 3, 4])
y = np.array([3, 4, 5, 6])

# addition
print(x + y)
print(np.add(x, y))

# subtraction
print(x - y)
print(np.subtract(x, y))

# multiplication
print(x * y)
print(np.multiply(x, y))

# division
print(x / y)
print(np.divide(x, y))

# in the above examples, you can use constants in place of the
                                variables

# arrays must be same shape (or broadcastable)
x = x.reshape(2, 2)
y = y.reshape(2, 2)
# you can perform all the same as above

# other functions
np.sqrt(x)
np.exp(x)
np.power(x, 2)

# statistical methods
x.mean()
x.mean(axis = 0)
x.mean(axis = 1)
x.std()
x.median()
x.max()
x.min()

# methods across all elements
x.sum()
```

```
x.sum(axis = 0)
x.sum(axis = 1)

# Broadcasting
X = np.arange(9).reshape(3, 3)
z = np.arange(3)
"""
X =
[[0 1 2]
[3 4 5]
[6 7 8]]

z =
[0 1 2]

X + z =
[[ 0  2  4]
[ 3  5  7]
[ 6  8 10]]
"""
# the example performs addition by column-wise broadcasting
# row-wise broadcasting is also possible, try using z.reshape(3,
                                    1)
```

9 Exercise: Creating ndarrays with Broadcasting

NumPy Mini-Project: Mean Normalization

In machine learning we use large amounts of data to train our models. Some machine learning algorithms may require that the data is normalized in order to work correctly. The idea of normalization, also known as feature scaling, is to ensure that all the data is on a similar scale, i.e. that all the data takes on a similar range of values. For example, we might have a dataset that has values between 0 and 5,000. By normalizing the data we can make the range of values be between 0 and 1.

In this lab, you will be performing a different kind of feature scaling known as mean normalization. Mean normalization will scale the data, but instead of making the values be between 0 and 1, it will distribute the values evenly in some small interval around zero. For example, if we have a dataset that has values between 0 and 5,000, after mean normalization the range of values will be distributed in some small range around 0, for example between -3 to 3. Because the range of values are distributed evenly around zero, this guarantees that the average (mean) of all elements will be zero. Therefore, when you perform mean normalization your data will not only be scaled but it will also have an average

of zero.

To Do:

1. Create a rank 2 ndarray of random integers between 0 and 5,000 (inclusive), with 1000 rows and 20 columns. This will simulate a large dataset with a wide range of values.
2. Normalize the array using mean normalization, where norm_col = (col - mean) / std.
3. Use broadcasting to calculate the the mean normalized version of the data.
4. Check results: the average of the mean normalized data should be close to zero, and the values should be evenly distributed around zero.
5. Split the mean normalized data into a training set (60%), a cross validation set (20%), and a test set (20%).

```
# Step 1: Create Array
import numpy as np

# Create a 1000 x 20 ndarray with random integers in the half-
                                open interval [0, 5001).
X = np.random.randint(1, 5001, size = (1000, 20))

# Step 2: Normalize Array using Mean Normalization (norm_col = (
                                col - mean) / std)
# mean is the average values of the column and std is the
                                standard deviation of the column

# Average of the values in each column of X
ave_cols = X.mean(axis = 0)

# Standard Deviation of the values in each column of X
std_cols = X.std(axis = 0)

# Step 3: Use Broadcasting to Calculate the Mean Normalized
                                Version of X
# Mean normalize X
X_norm = (X - ave_cols) / std_cols

# Step 4: Check results
# Average of X_norm values should be close to zero, and values
                                should be evenly distributed
                                around zero
# Print the average of all the values of X_norm
print(X_norm.mean())

# Print the average of the minimum value in each column of X_norm
print(X_norm.min(axis = 0).mean())

# Print the average of the maximum value in each column of X_norm
print(X_norm.max(axis = 0).mean())
```

```
# Step 5: Split dataset into
# Traning Set (60%)
# Cross Validation Set (20%)
# Test Set (20%)

# First, we'll create a random row indices vector
# Create a rank 1 ndarray that contains a random permutation of
                                the row indices of `X_norm`
row_indices = np.random.permutation(X_norm.shape[0])

# Second, we'll create some variables for splitting the set
total_rows = X_norm.shape[0]
train_rows = round(total_rows * .6)
cross_rows = round(total_rows * .2)
test_rows = total_rows - train_rows - cross_rows

train_indices = row_indices[:train_rows]
cross_indices = row_indices[train_rows:cross_rows + train_rows]
test_indices = row_indices[cross_rows + train_rows:]

# Third, we'll create the sets by slicing the data using the
                                permutated indices
# Create a Training Set
X_train = X_norm[train_indices]

# Create a Cross Validation Set
X_crossVal = X_norm[cross_indices]

# Create a Test Set
X_test = X_norm[test_indices]
```