Assignment-1

NUMPY and PANDAS Documentation

-Gursimar Singh Bedi

# Installation

We can use pip in the command line client or in our Jupyter notebook to install both Pandas and Numpy.

**In [1]:!pip install** **numpy**

**In [2]:!pip install** **pandas**

# Import Statements

The import statements are to be included at the beginning of every project which requires the libraries. The as keyword is used to give aliases or short forms. They are optional and just for convenience of the user.

**In [1]: import** **numpy** **as** **np**

**In [2]: import** **pandas** **as** **pd**

# Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

## Numpy arrays

A numpy array is a grid of values of the same type. It is 0 indexed. The number of **dimensions is the rank** of the array. Numpy arrays always have a fixed size and (unlike normal series) facilitate advanced mathematical operations on enormous sets of data.

### Creation and Initialization functions

**In [1]: a=np.array([1,2,3])** #1d numpy array

**In [2]: b=np.array([[1,2,3],[2,3,4]])** #2d numpy array

### Shape

The shape of an array is a tuple of integers giving the size of the array along each dimension.

**In [1]: a.shape()** #prints(3,)

**In [2]: b.shape()** #prints(2,3)

### Type of Object

**In [1]: np.dtype(a)** #prints np.int32

### Creating Different Types of Arrays

In mathematics we have predefined terms for arrays which come in handy. Numpy supports the following types.

#### Zeros

Creates an array of the given shape with only zero values.

**In [1]: a=np.zeros((2,2))** #Prints [[0. 0.]

**In [2]: print(a)** # [0. 0.]]

#### Ones

Creates an array of the given shape with ones.

**In [1]: b=np.ones((1,2))** #Prints [[1. 1.]]

**In [2]: print(b)**

#### Empty

Return a new array of given shape and type, without initializing entries.

**In [1]: c=np.empty((2,1))** #Prints [[-9.74499359e+001]

**In [2]: print(c)** # [2.13182611e-314]]

##### Eye (identity matrix)

Return a 2-D array with ones on the diagonal and zeros elsewhere

**In [1]: i=np.eye((2,2))** #Prints [[1. 0.]

**In [2]: print(i)** # [0. 1.]]

#### Arange

Returns a sequence in a form of an array. Is particularly helpful in creation of AP.

**Syntax: np.arange( starting\_value,ending\_value,difference)**

**In [1]: i=np.arange(0,10,2)** #Prints [0. 2. 4. 6. 8.]

### Scalar Operation on Numpy Array

**In [1]: i=np.arange(0,10,2)** #Prints [0. 2. 4. 6. 8.]

**In [2]: i\*i** #Multiplies each index of 1st array with 2nd array [0. 4. 16. 36. 64.]

**In [3]: i\*\*2** #Exponent with every index array [0. 4. 16. 36. 64.]

**In [4]: i-i** #Prints [0. 0. 0. 0. 0.]

**In [5]: i+i** #Prints [0. 4. 8. 12. 16.]

### Indexing

Indices are used to access elements in the numpy array. To access an individual element in an array we need square brackets equivalent to the number of ranks.

**In [1]: i1=np.arange(0,10,2)** #Prints [0. 2. 4. 6. 8.]

**In [2]: i2=np.array([[1,2],[3,4]])**

**Accessing Individual Elements**

**In [3]: i1[0]** #first element:0

**In [4]: i2[0][0]** #first element:1

**Accessing Range through Slicing**

**In [5]: i1[0:3]** # [0 2 4 6]

**In [6]: i2[:,1]** # [2,4]

**Accessing Row**

**In [7]: i2[0]** # [1 2]

### Slicing

Slicing is the process of getting a range of elements using a colon. If no number is written on any side of colon, then all numbers are selected. Comma separated values in slicing can be used to access values from nd-arrays.

**If an array is made out of slicing it points to memory location so any changes made in that array result in the change being reflected in our original array as well.**

**In [1]: i1=np.arange(0,10,2)** #Prints [0. 2. 4. 6. 8.]

**In [2]: i2= i1[0:3]** # [0 2 4 6]

**In [3]: i2[0]=5**

**In [4]: i2[1]=5**

**In [5]: i2** # [5 5 4 6]

**In [6]: i1** # [5 5 4 6 8] Values in original array changes as well.

## Copy Function

To avoid the first situation we use the copy function. It duplicates the array and then array manipulation in the duplicated array doesn’t affect the original array.

**In [1]: i1= np.arange(0,10,2)** #Prints [0. 2. 4. 6. 8.]

**In [2]: i2= np.copy(i1[0:3])** # [0 2 4 6]

**In [3]: i2[0]=5**

**In [4]: i2[1]=5**

**In [5]: i2** # [5 5 4 6]

**In [6]: i1** # [0 2 4 6 8]

## Accessing rows using list of index values

Sometimes we require to access individual elements from the arrays based on some condition. We can use a list to do so.

**In [1]: arr= np.arange(0,10,2)** #Prints [0. 2. 4. 6. 8.]

**In [2]: list= [True, False, True, False, False]**

**In [3]: arr[list]** #Prints [0. 4.]

## Premium array operation

**In [1]: i=np.array([1,4,9])**

**In [2]: np.sqrt(i)** #Prints square root [1. 2. 3.]

**In [3]: np.exp(i)** #Prints exponential i.e e^value

**In [4]: ir=np.array([9,4,1])**

**In [5]: np.add(i,ir)** #Adds the arrays [10. 8. 10.]

**In [6]: np.maximum(i,ir)** #Prints maximum value [9. 4. 9.]

## Random Values

For testing purposes we require random values in an array. We can do it using the following command.

**In [1]: i=np.random.rand(2,2)** #Random array of the given shape

## Saving Numpy Arrays

Jupyter Notebook contains a kernel which holds the memory of all the commands that we ran until that point. Once we close the Notebook the kernel is turned off. Hence we can our created arrays which may take hours to recreate. Here comes the importance of saving our arrays.

### Saving a Single Array

**In [1]: i=np.array([1,4,9])**

**In [2]: np.save(‘final\_array’,i)** #Saves array at jupyter notebook file location

Once we are back we can just load the array again

**In [1]: ir=load(‘final\_array.npy’)**

**In [2]: ir** #Prints [1. 4. 9.]

### Saving Multiple Arrays

Much like a zip file a .npz file is a compressed format to save multiple arrays. Each array is given a title to access it in the future.

**In [1]: i1=np.array([1,4,9])**

**In [2]: i2=np.array([2,4,6])**

**In [3]: np.savez(‘final\_arrays.npz’,a=i1,b=i2)**

Once we are back we can just load the array again

**In [1]: i3=load(‘final\_array.npz’)**

**In [2]: i3[‘a’]** #Prints [1. 4. 9.]

### Saving arrays into text files

Although npy and npz formats are helpful for saving arrays, unfortunately we cannot read the arrays in this format. Numpy gives us the ability to save our arrays in a text file format. Here comes the use of a delimiter i.e. a separator which helps identify each element of the array. The only issue with saving files in such format is that it adds floating points to all values.

**In [1]: i1=np.array([1,4,9])**

**In [2]: i2=np.array([2,4,6])**

**In [3]: np.savetxt(‘final\_arrays’,i1,i2,delimiter=”,”)**

Once we are back we can just load the array again

**In [1]: i3=loadtxt(‘final\_array.txt’)**

**In [2]: i3** #Prints [1. 4. 9. 3. 4. 6.]

# Pandas

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

## Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index. We can create a series from a numpy array as well

## Data Frames

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object.

## Creating a Series

By default a series is 0 indexed but we can give our own indices to locate values

**In [1]: s=pd.series([1,2,3])**

**In [2]: s.values** #Prints [1 2 3]

**In [3]: s.index.values** #Prints [0 1 2]

**In [4]: s[0]** #Prints 1

### Custom indexed series

To give custom indices we need to declare them at the time of initialization.

**In [1]: s=pd.series([1,2,3],index=[‘First’, ’Second’, ’Third’])**

**In [2]: s.values** #Prints [1 2 3]

**In [3]: s.index.values** #Prints [‘First’, ’Second’, ’Third’]

**In [4]: s[‘First’]** #Prints 1

### Accessing Array Elements based on Conditions

Conditions can be given in square bracket to get a sub-array.

**In [1]: s=pd.series([100,20,1,2,3,4])**

**In [2]: s[s<5].values** #Prints [1 2 3 4] ie al values less than 5

### Check if Element is in Pandas Series

**In [1]: s=pd.series([100,20,1,2,3,4])**

**In [2]: s.isin(1)** #true as 1 is in series

**In [3]: s.isin(55)** #false as 55 is not in series

## Dictionaries

A dictionary is as simple as placing items inside curly braces {} separated by commas. An item has a key and a corresponding value that is expressed as a pair (key: value). While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

**In [1]: s=pd.series([1,2,3],index=[‘First’, ’Second’, ’Third’])**

**In [2]: dict=pd.to\_dict(s)** #Prints {‘First’:1, ’Second’:2, ’Third’:3}

#The indices turns into keys

**In [3]: s1=pd.DataFrame(dict)** #Converts back to dataframe

**First 1**

**Second 2**

**Third 3**

## Giving Column names

**In [1]: s=pd.series([1000,2000,3000],index=[‘BMW’, ’AUDI’, ’Ferrari’])**

**In [2]: s.names=”Cost”**

**In [3]: s.index.names=”Car”**

**Car Cost**

**BMW 1000**

**AUDI 2000**

**Ferrari 3000**

## Operations in series and DataFrames

Operations work the same as they work in numpy. Series replicate 1-d array and df replicate 2d arrays.

## Accessing Rows in DataFrames

#### Return the first five rows of a df

**In [1]: df.head()**

#### Return Last Five rows of a df

**In [1]: df.tail()**

#### Return column names of a df

**In [1]: df.columns()**

#### Returns a column/subset of columns

**In [1]: df[column\_name]**

**In [1]: df[[list of comma serpeated column\_names]]**

## Deleting a Column

**In [1]: pd.del(df[col\_name])**

## Renaming Indices in a series

Once assigned, one cannot change indices of a series/ a dataframe. We can use the rename to duplicate an object with renamed indices. Assigning the new object to previous object can be considered equal to changing the indices in theory.

**In [1]: s=pd.series([1000,2000,3000],index=[‘BMW’, ’AUDI’, ’Ferrari’])**

**In [2]: s.names=”Cost”**

**In [3]: s.index.names=”Car”**

**Car Cost**

**BMW 1000**

**AUDI 2000**

**Ferrari 3000**

**In [4]: s.rename(index ={”BMW”:”Honda”})**

**Car Cost**

**Honda 1000**

**AUDI 2000**

**Ferrari 3000**

**In [1]: df**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**2 Ferrari 30000**

**In [2]: df.reindex([0,1,3])**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**3 Nan Nan**

**In [3]: df.reindex([0,1,3],fill\_value=10)**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**3 10 10**

**In [4]: df.reindex([0,1,2,3],method=’ffil’)**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**2 AUDI 20000**

**3 10 10**

### Forward Fill

It is used to fill the missing value in the dataframe. ‘ffill’ stands for ‘forward fill’ and will propagate last valid observation forward.

## Dropping values from series/DF

**In [1]: df**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**2 Ferrari 30000**

### Dropping Rows

**In [2]: df.drop([0,1])**

**Car Cost**

**2 Ferrari 30000**

### Dropping Columns

**In [2]: df.drop(1,axis=1)**

**Car**

**0 Honda**

**1 AUDI**

**2 Ferrari**

## Handling Null/Nan Values

### Checking for null values

**In [1]: df**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**3 Nan Nan**

**In [2]: df.isnull()** #returns true for row indexed 3

### Dropping null values

The issue with dropping null values is that even if there is one null value it deletes the entire row and we lose data.

**In [1]: df**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**3 Nan Nan**

**In [2]: df.dropna()**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**In [3]: df\_new**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**3 Apple Nan**

**In [3]: df\_new.dropna(how=”all”) # drops a row only when all values are missing**

**Car Cost**

**0 Honda 10000**

**1 AUDI 20000**

**3 Apple Nan**