Pandas

## Introduction to Pandas

* Pandas is an open-source library built on top of Numpy which allows for fast data cleaning, preparation and analysis
* You can install the library in command prompt using “conda install pandas” or “pip install pandas” for conda and general distribution of python respectively.
* The following topics will be covered in the course: series, data frames; missing data; groupby, merging, joining, and concatenating; operations; data input and output.

## Series

* The primary differences between NumPy and Pandas series are: pandas series has axis labelling i.e., can be indexed by a label and can hold almost any data type including strings, numbers, lists, dictionaries, lists, numpy arrays, python built in functions etc.
* You can create a series with pandas.Series(data, label) where the label can be used to call a data element.
* The data can be a list, numpy array, dictionary etc.
* When a dictionary is used, Pandas Series directly uses the keys as label and the values as data point
* Series automatically indexes data points with index numbering if the labels was not specified
* To grab a data point from a pandas series is similar to grabbing from a dictionary or numpy array and uses the line Seriesname[index]
* Note that when performing math operations in a pandas or numpy array, it converts the data points to floats automatically to preserve as much information as possible.

## Data frames - 1

* To ensure you always get the same set of random numbers, from the numpy library, use np.random.seed(101)
* To create a data frame, use pd.DataFrame(a,b,c) where a=data point, b=labels/index/rows/’y index’ and c=columns/’x index’ respectively.
* Each column is just a pandas series whereas a data frame is a bunch of series that share an index.
* To grab a column, use dataframename[a] where a = column label for a single column and dataframename[[a, b, c]] where the list of indices [a, b, c etc.] specify the desired columns.
* You can add a new column by specifying it name and using arithmetic on existing columns to add a new column with dataframename[t] = v where t = new column name and v = arithmetic for computing new column.

Note: columns can be overwritten just like a dictionary

* You can drop a column using dataframename.drop(a,b) where a = column name and b (axis) = 1 for columns or b = 0 for rows
* Dropping a column doesn’t happen in place which means if I called the data frame in future, it will still contain the dropped column
* To ensure dropped columns are done in place, you have to specify in place as True e.g., dataframename.drop(column name, axis = 1, inplace=True)
* You can also drop a row using dataframename.drop(row index, axis=0)
* To select a row, you can use dataframename.loc[index name] or dataframename.iloc[index number] for label index and numerical index respectively.
* To grab an element of the dataframe, you can use dataframename[a,b] where a = row index and b = column index respectively.
* To grab a sub set of the dataframe, you can use dataframename[[a, b, c…],[1,2,3..]] where a, b, c etc. is a list of the index of desired rows and 1, 2, 3 etc. a list of index of desired columns respectively.

Note: you cannot use slice notations to grab data points from a data frame with .loc.

## Data frames - 2

* Conditional selection of data from a data frame can be done with comparism operators e.g., df>0 which returns a data frame of Booleans for when this is true or false.
* You can instead cast the result of this conditional selection to show the values where the condition is True with df[df>0] which returns the data frame as the numbers that are true and NaN for where it is false
* You can use conditional statements to grab values of only rows where a conditional statement is met by a given column e.g., df[df[‘a’]>0] where ‘a’ is the column where the condition should be checked against. This returns a data frame of rows where the stated condition is true and ignores any row where the condition is false
* You can select some columns based on a condition in a column by stacking the commands on the resulting conditional command e.g., df[df[4]>0] [‘B’] or df[df[4]>0] [[‘B’,‘D’]] for a single column or two or more columns respectively. Note that for more than one column, the column index is passed in as a list of the indices.
* You can do conditional selection based on more than one condition with df[(df[4]>0]) & (df[4]>1])] or df[(df[4]>0]) | (df[4]>0])] for ‘and’ and ‘or’ respectively.
* Note that to join conditions, python’s global ‘and’ and ‘or’ commands return an error but are replaced with ‘&’ and ‘|’ respectively.
* You can reset the row index of a data frame back to default numbering with df.reset\_index(). This converts your present specified index to a new column of data in your data frame.
* The reset index operation does not occur in place, to make it in place, specify with df.reset\_index(inplace=True)
* You can add a new column to a data frame by attaching a matching table with df[column name] = list name
* You can use a column in your data frame as the row index with df.set\_index(column name). doesn’t occur in place except specified

## Data frames – 3

* MultiIndex is a pandas method that can take a data set such as a numpy array, tuples, data frame etc. and make a multi index off of it.
* Multi-level or higher index data frames are called from outside inwards e.g., df.loc[level1].loc[level2] etc.
* You can name indexes with the df.index.names = [a,b] where a and b are names corresponding to the different levels of index in order. You can have as many of the indexes as you do like.
* You can grab any data point from any level of indexing from outside in. e.g., df.loc[1st level].loc[2nd level] etc.
* You can skip levels to grab any data set using cross section with df.xs(a,level=b) where a and b are the index and level name in multi index respectively.

## Missing data

* A data frame can be created from a dictionary with pd.DataFrame(dictionary name)
* You can drop any column or row with at least one Nan(null value) in it with df.dropna(axis=1) and df.dropna() respectively.
* You can set a threshold of how many Nan values are required in an index before being dropped with df.dropna(thresh=x), where x is the integer value of minimum required number of NaN values to drop a row.
* You can fill missing values (NaN) with any value or the mean of the column using df[Column title].fillna(value=a) where a = a function/string/number etc.

## Group by

* Group by allows you group rows together and call aggregate functions.
* You can group data point using df.groupby(‘a’) where a is the column you are grouping by. You have to save this to a variable and call operations off it later on e.g., y = df.groupby()
* You can call operations such as standard deviation, sum, mean etc. off it with y.mean/sum/std()
* You can call data from an aggregate using the grouping variable e.g., y.mean().loc[a] where a is the specific group you want.
* In practice, you don’t often get too break these steps down, instead, you make a one liner such as df.groupby(a).sum().loc[b] where a and b are grouping column and specific group. Note, this is an example of sum and there are others.
* Other useful aggregates include count, max, min, etc
* You can easily get a bunch of information about a data frame by calling ‘.describe()’ with the group by method e.g., df.groupby().describe(). It gives you the mean, standard deviation, min, max, count, quartile etc
* You can transpose the resulting data frame from ‘.describe()’ if you prefer by calling ‘.transpose()’ on it with df.groupby().describe().transpose()
* You can also call single column from the resulting data frame with df.groupby().describe().transpose()[a] where ‘a’ is the desired column

## Merging Joining and Concatenating

* Concatenating simply glues the different data frames together along the specified axis. The axis must be equal in length. It stacks each data frame on top of each other.
* You can concatenate with pd.concat([a]) where a is a list of the data frames to be concatenated with default axis = 0, you can concatenate along the columns by setting axis=1 i.e. pd.concat([a, b], axis=1) where a b etc. are names of the data frames to be merged.
* Merge allows you to merge data frames together and create a single column for the similar columns in the different data frames
* You can merge with pd.merge(a, b, how=x, on=y) where a and b are the data frames, x is the manner of merging- inner(default) or outer/right/left, y is the column to be unified into one in the resulting data frame.
* Note that y above must have the same column title
* You can merge on multiple columns by passing a list e.g., pd.merge(a, b, how=x, on=[y, z]) where y and z are the columns to merge on respectively.
* Concatenating is vertical, merging is horizontal,

Note: I don’t fully understand merging yet especially the ‘how’ command and how it affects the arrangement. Read documentation and learn excel afterwards.

* Join is used to merge data frames across rows. This means it can be used to glue data that is not necessarily homogenous in the column labelling.
* You can use the command a.join(b) where a and b are the respective data frames.
* Join can also take a ‘how’ instruction.

## Operations

* You can return the first ‘n’ number of rows using df.head(n) where n is the number of rows.
* You can return all the unique values in a column using the df[column name].unique()
* You can return the number of unique values in a column using the df[column name].nunique() or you could use the ‘len’ command on the df[column name].unique() with len(df[column name].unique())
* You can return the number of times every unique data frame element on the column appear with the ‘value\_counts’ command e.g., df[‘col2’].value\_counts()
* df7[df7['col1']>2], df7[(df7['col1']>2) & (df7['col2']>500)] and df7[(df7['col1']>2) | (df7['col2']>500)] are different forms of conditional selection respectively.
* You can broadcast your own function or some other python commands to a column of data in a data frame using df[column name].apply(a) where ‘a’ is the function or method.
* You can also use the ‘apply’ method with a lambda expression by replacing ‘a’ with a lambda expression. This removes the need to define a full function.
* df.drop(column name, axis=1, inplace) drops a column based on the specified column name and whether its in place or otherwise.
* df7.columns and df7.index returns column titles and index/row titles for all the columns
* you can sort the values of a data frame using a column or index with df.sort\_values(by=a, axis) where ‘a’ is the column or df7.sort\_index() to sort by index.
* df.isnull() returns Boolean for each cell in a data frame indicating if there is a null value(NaN) or not. False implies the cell is not a null value. You also have the ‘isin’ and ‘isna’ to check if every element of a data frame is contained in values and whether there is a missing value in a data frame.
* You can create a pivot table from a dt frame with df.pivot\_table(values, index, columns). Note that using a single index can sometimes result in the data being aggregated in the returned data frame and the default aggregate is mean but can be changed to any type of aggregate such as sum etc.

## Data input and output

CSV files

* You can read a csv file with pd.read\_csv(a) where ‘a’ is file path or file name if it is in the same folder with the notebook file.
* You can write into a csv file by first assigning the file into a data frame when reading with data frame name = pd.read\_csv(a)
* You can write (create) into a csv file with assignedname.to\_csv(‘input’)
* It is usual to set the index to False to avoid the input distorting the index (creating a new index and assigning the original as a column by using assignedname.to\_csv(‘input’, index=False) instead of just assignedname.to\_csv(‘input’).

### Excel files

* Pandas can only import the data and not formulas, macros, images etc. due to this, using pd.read\_excel() may cause pandas to crash.
* You can read an excel file with pd.read\_excel(a, b) where ‘a’ is the file name and b is the sheet name.
* To write data into a new excel file, use df.to\_excel(data frame name, sheet name)

### Html

* You can read from a html address using pd.read\_html(html link)

### SQL

* It is not the best way to read a SQL file due partially to the different type of sequel engines such as MySQL, SQLite etc.
* To create a temporary SQL data in memory, ‘from sqlalchemy import create\_engine’, then, use a create\_engine(‘sqlite:///:memory:’). This is for SQLite
* To write into the temporary file, use df.to\_sql(a, b) where a=data, and b = connection
* You can assign this to a variable with pd.read\_sql(data,con) and read it by calling the variable.

### Notes

* You can check the data at the top of the data frame with df.head(a) where a is number of rows from the top.
* You can use df.info() to get a concise summary of a data frame column wise such as number of entries and NaN in every column.