Pandas

What will you learn?

1. Introduction to Pandas
2. Reading the Data
3. **Functionalities of Pandas** : Creation, Viewing, Editing
4. Manipulating Data
5. Handling NaN
6. **Handling Duplicates** : Row Index, Column Names
7. Handling String Data

Pandas is an open source library which provides high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Pandas has a lot of functions that will help in reading and writing data and also for data manipulation. Thus we will be using pandas throughout the course.

Pandas behave like an excel file.

Lets import pandas and read some data.

In [ ]:
```
#Import Pandas
import pandas as pd
```

Reading Data

We will use **read_csv()** function. It reads a comma-separated values (csv) file into DataFrame.

In [ ]:
```
#Loading data with read_csv() function. Here we are providing path to the csv
file.
#If the file is in your system you can provide its path as well.
iris = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/
iris/iris.data")
```

In [ ]:
```
type(iris)
```

Pandas Dataframes

DataFrame is an object for data manipulation. You can think of it as a 2D tabular structure, where every row is a dataset entry and columns represents features of data.

In [ ]:
```
iris
```

By default, the first row of the csv file has been used as column names. We will soon see how to fix that.

Creating copy of DataFrame

In [ ]:
```
df = iris
## Above statement simply makes df refer to the data frame object that iris is
referring to.
## So now both iris and df refer to the same dataframe object and any changes
done via one will reflect in other.
## So effectively this is not creating another dataframe object.
```

If we wish to create a copy then we will use **copy()** function for that

In [ ]:
```
df = iris.copy()
```

In [ ]:
```
df.shape
```

As you can see, we have 149 rows and 5 columns. But actually, this should have been 150 rows, as we already know, the Iris Dataset has information of 3 different types of flower, 50 each. This happened because the first row was taken as the column name. To fix this, we do the following:

In [ ]:
```
#Ignoring header -> If you don't want first row to be treated as a header, you
can set header = None
iris = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/
iris/iris.data", header=None)
iris
```

In [ ]:
```
df = iris.copy()
df.shape
```

To see the datatypes of each column we do the following:

In [ ]:
```
df.dtypes
```

Currently, our columns have no names.

In [ ]:
```
df.columns
```

To give them a name, we simply change the value of df.columns

In [ ]:
```
df.columns = ['sl', 'sw', 'pl', 'pw', 'flower_type']
df
```

In [ ]:
```
df.dtypes
```

We may get a quick analysis of our data using **describe()**

In [ ]:
```
df.describe()
```

Some Basic Functionalties

Viewing the DataFrame

We have the **head()** and **tail()** function for viewing the dataframe.

head()

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

By default, value of n = 5.

In [ ]:
```
df.head()
```

In [ ]:
```
df.head(10)
```

tail()

This function returns the last n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

By default, value of n = 5.

In [ ]:
```
df.tail()
```

In [ ]:
```
df.tail(11)
```

Accessing Data

Sometimes, we may want to look at a single column from the DataFrame. This can be done simply as:

In [ ]:
```
## Viewing sl column
df.sl
```

and

In [ ]:
```
df['sl']
```

Checking for NULL values

In [ ]:
```
df.isnull()
```

In [ ]:
```
# To get a direct overview
df.isnull().sum()
```

Selection

iloc[]

We can use the **iloc[ ]** function to access values in dataframe.

It is a purely integer-location based indexing for selection by position. iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

1. An integer, e.g. 5.
2. A list or array of integers, e.g. [4, 3, 0].
3. A slice object with ints, e.g. 1:7.
4. A boolean array.

In [ ]:
```
df.iloc[1:4, 2:4]
```

loc[ ]

This accesses a group of rows and columns by label(s) or a boolean array.

.loc[ ] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

1. A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
2. A list or array of labels, e.g. ['a', 'b', 'c'].
3. A slice object with labels, e.g. 'a':'f'.
4. A boolean array of the same length as the axis being sliced, e.g. [True, False, True].

In [ ]:
```
df1 = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
     index=['cobra', 'viper', 'sidewinder'],
     columns=['max_speed', 'shield'])
df1
```

In [ ]:
```
df1.loc['viper']
```

In [ ]:
```
df1.loc[['viper', 'sidewinder']]
```

DataFrame from Dictionary

In [ ]:
```
mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
          {'a': 100, 'b': 200, 'c': 300, 'd': 400},
          {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
df1 = pd.DataFrame(mydict)
df1
```

Manipulating data

Deletion of data

drop()

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

It returns us a DataFrame without the removed index or column labels, or None if inplace=True.

In [ ]:
```
df.head()
```

In [ ]:
```
a = df.drop(0)
a.head()
```

To actually change the data in the original dataframe, we use the parameter 'inplace = True'

In [ ]:
```
df.head()
```

In [ ]:
```
df.drop(0, inplace = True)
df.head()
```

Let's try to do this again

In [ ]:
```
df.drop(0, inplace = True)   #Error Generated
df.head()
```

The reason for this is, after dropping 0, the indexing did not change automatically. Now, the labels do not begin from 0, but 1.

As we learnt in the definition, we are removing rows by their labels. To remove rows by their indices, we may do the following:

In [ ]:
```
df.drop(df.index[0], inplace = True)
df.head()
```

In [ ]:
```
df.drop(df.index[3], inplace = True)   ## Label 5 removed
df.head()
```

We may also remove many labels in one go.

In [ ]:
```
df.drop(df.index[[3, 4]], inplace = True)   ## Label 6, 7 removed
df.head()
```

In a similar manner, we may remove columns.

In [ ]:
```
df.drop('sl')   ## Error Generated
```

An error is generated because the drop function is currently looking for a row with label 'sl'. We need to change the axis.

In [ ]:
```
df.drop('sl', axis = 1)
```

Conditional Insights

We may use concept of boolean indexing in DataFrame to access a particular type of data, and draw inferenced from it.

In [ ]:
```
df
```

Lets try to gain insights of data correspondign to Iris-virginica.

In [ ]:
```
df[df.flower_type == 'Iris-virginica'].describe()
```

Addition of data

loc()

In [ ]:
```
df.loc[0] = [1, 2, 3, 4, 'Iris-virginica']
df.tail()
```

We may directly create new columns also according to our needs.

In [ ]:
```
df["diff_of_sl_sw"] = df['sl'] - df['sw']
df.head()
```

In [ ]:
```
df.drop('diff_of_sl_sw', axis = 1, inplace = True)
```

Reset Index

After removing certain rows, the order of indices got changed. We can reset it using the **reset_index()** function.

In [ ]:
```
df.reset_index()
```

But this has created an additional column with old indices. To avoid that, we do:

In [ ]:
```
df.reset_index(drop = True)
```

Handling NaN

Values considered “missing”

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python None will arise and we wish to also consider that “missing” or “not available” or “NA”.

To make detecting missing values easier (and across different array dtypes), pandas provides the **isna()** and **notna()** functions, which are also methods on Series and DataFrame objects.

Because NaN is a float, a column of integers with even one missing values is cast to floating-point dtype

NaN values can create inaccuracies in our estimations and calculations. There are two ways we can handle NaN:

1. we either remove them,
2. or we fill them.

Our current data does not have any NaN values, so we will create some.

In [ ]:
```
import numpy as np
df = iris.copy()
df.columns = ['sl', 'sw', 'pl', 'pw', 'flower_type']
```

In [ ]:
```
df.iloc[2:4, 1:3] = np.nan
df.head()
```

In [ ]:
```
df.describe()
```

Dropping NaN

dropna() : This will remove the row or column entries with NaN values.

In [ ]:
```
df.dropna(inplace = True)  ## Remove NaN inside df only
df.reset_index(drop = True, inplace = True)   ## Reset the indices
```

In [ ]:
```
df.head()
```

As you may observe, we have removed the row with NaN. If we want to remove the column, we shall use 'axis' parameter.

Filling NaN

fillna() : You can also fill NaN using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill.

Generally we fill the NaN values with the mean, but depending on the type of data, and your own analysis, you may decide to will NaN in some other way.

In [ ]:
```
df.iloc[2:4, 1:3] = np.nan
df.head()
```

In [ ]:
```
df.sw.fillna(df.sw.mean(), inplace = True)
df.pl.fillna(df.pl.mean(), inplace = True)
df.head()
```

Note: Since all the NaN values belonged to 'Iris-setosa', a better value to fill NaN's would have been the mean of those values of 'sw', where flower type is Iris-setosa.

In [ ]:
```
df.iloc[2:4, 1:3] = np.nan
df.head()
```

In [ ]:
```
df_setosa = df[df.flower_type == 'Iris-setosa']
df.sw.fillna(df_setosa.sw.mean(), inplace = True)
df.pl.fillna(df_setosa.pl.mean(), inplace = True)
df.head()
```

Duplicate Labels

Index objects are not required to be unique; you can have duplicate row or column labels.

But one of pandas' roles is to clean messy, real-world data before it goes to some downstream system. And real-world data has duplicates, even in fields that are supposed to be unique.

Lets see how duplicate labels change the behavior of certain operations, and how prevent duplicates from arising during operations, or to detect them if they do.

Consequences of Duplicate Labels

Some pandas methods (Series.reindex() for example) just don't work with duplicates present. The output can't be determined, and so pandas raises.

Other methods, like indexing, can give very surprising results. Typically indexing with a scalar will reduce dimensionality. Slicing a DataFrame with a scalar will return a Series. Slicing a Series with a scalar will return a scalar. But with duplicates, this isn't the case.

In [ ]:
```
df1 = pd.DataFrame([[0, 1, 2], [3, 4, 5]], columns=["A", "A", "B"])
df1
```

We have duplicates in the columns. If we slice 'B', we get back a Series

In [ ]:
```
print(df1["B"])  # a series
type(df1["B"])
```

But slicing 'A' returns a DataFrame

In [ ]:
```
print(df1["A"]) # a DataFrame
type(df1["A"])
```

This applies to row labels as well.

In [ ]:
```
df2 = pd.DataFrame({"A": [0, 1, 2]}, index=["a", "a", "b"])
df2
```

In [ ]:
```
df2.loc["b", "A"]  # a scalar
```

In [ ]:
```
df2.loc["a", "A"]  # a Series
```

Duplicate Label Detection

You can check whether an Index (storing the row or column labels) is unique with **Index.is_unique**:

In [ ]:
```
df2
```

In [ ]:
```
df2.index.is_unique
```

In [ ]:
```
df2.columns.is_unique
```

Index.duplicated() will return a boolean ndarray indicating whether a label is repeated.

In [ ]:
```
df2.index.duplicated()
```

Handling Strings in Data

Our algorithms can make calculations over numerical data. String data is very hard to compute quantitaviely.

It wont make sense to ignore string data. For example, if a dataset is to evaluate shopping habits, and we have a column for gender with categories as 'male' and 'female', we cannot just ignore this, as the habits of both the gender will be very different from each other.

So, to handle such cases, we convert the string data to numerical data.

In [ ]:
```
df
```

Lets create a dummy column to understand the process.

In [ ]:
```
df['Gender'] = 'Female'
df.iloc[0:10, 5] = 'Male'
df
```

In [ ]:
```
def func(s):
  if s == 'Male':
    return 0
  else:
    return 1

df['Sex'] = df.Gender.apply(func)
del df['Gender']
df
```

Now, we may apply algorithms which take into consideration the 'Sex' column too.