Pandas Notes in CDE

an open-source software library built on top of Python specifically for data manipulation and analysis, Pandas offers data structure and operations for powerful, flexible, and easy-to-use data analysis and manipulation.

# Making DF

You can make dataframe using pd.DataFrame(). These parentheses should include list, or 2d array or anything that can be changed into a dataframe.

# Head () & Tail ()

This method prints first five rows of the dataframe for information purpose. If you specify like df.head(3) so it’ll print the required rows. Tail prints the last 5 rows of dataframe.

# Column names and index:

While making a df, pandas will automatically take the first row as column names. But if you want to name the column differently you can give column parameter the names as list. Same can happen with the index. For example:

pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], columns=["first", "second", "third"], index=["a","b","c"])

since the df is an object, so we can reach its attributes like columns and index. And we will get them as a list. The following code can be used for this purpose:

df.columns

df.columns.tolist()

the same code works for index.

# Info and describe:

To get general information about the df info() method can be used and for statistical information we can use describe() method.

# Shape and unique values:

Shape() will tell about the number of rows and columns in the dataframe and unique() returns an array of unique values in a Series or DataFrame column, while nunique() returns the count of unique values.

# Reading csv files:

For reading csv file and making df from it we can use read\_csv method. pd.read\_csv() automatically creates a DataFrame from the CSV file you provide. It reads the file and converts it into a pandas DataFrame object.

# Sample()

If you want to get random rows from dataframe so you can use sample() and input a number of rows you want to get In result.

# Loc()

Loc can help to get slicing or specific rows from dataframe. The following syntax can be used for slicing:

Df.loc(rows,columns)

If you ignore putting column or rows so it’ll consider taking data from all.

For single row df1.loc[10]

For range of rows df1.loc[5:10]

For range of rows with single column df1.loc[5:10, "Name"]

For range of rows with several columns df1.loc[5:10, ["Name","NOC"]]

For all rows but several columns df1.loc[:, ["Name","NOC"]]

# Iloc()

Its doing the same thing like loc() but the difference is that it needs index of columns instead of names.

df1.iloc[:,[1,3]]

# at and iat():

I you want to arrive at a specific value you can use at and Iat. Loc and iloc can also be used but they’re specifically for ranges instead of single values. At and iat are same as loc and iloc.

# Targeting column:

If you want to target one specific column so you can do it in two ways:

Df.column\_name

Df[“column\_name”]

But if you’re targeting multiple column then only the second way can be used.

# Sort\_Values():

Values of dataframe can be sorted by using the given function but one column name should be given as parameter so based on that folder the values can be sorted. If you wanna give several column names so you can give them in square brackets.

df1.sort\_values(["Age","ID"])

# conditional targeting:

if you want to get data from specific columns with specific conditions you can use the following two ways:

df1[(df1["Age"]<=13) & (df1["Year"]>1999)][["Name","Year","NOC"]]

df1.loc[df1["Age"]<=15, ["Name","Age", "City","Sport","Year"]]

# specific word in a column:

if you want to search all rows with a specific word in a column so you can use the following code:

df1[(df1["Name"].str.contains("Khan")) & (df1["NOC"].str.contains("PAK"))]

here I am search that gimme all columns where Name column contains Khan and noc column contains PAK.

The following code will also work for this purpose:

df1[df1["NOC"].isin(["USA"])]

df1[df1["NOC"].isin(["USA", "GBR"])]

the same result can we get from the following code:

df1.query('NOC == "USA"')

# searching by starting words:

for this purpose the following words of codes will work:

df1[df1["Name"].str.startswith("Akh")]

# Value\_counts():

To find which value is repeated how many times in a column.

df["Sex"].value\_counts()

# drop()

if you want to delete a row so write its index in the parenthesis and if you want to delete a specific column so write column para meter and give them name of that specific column.

df1.drop(0)

df1.drop(columns=["ID"])

# inplace=True

it’s a powerful parameter in pandas. The default value is false, but if the code is run so itll return a copy of the original dataframe and if the value is set to True so it’ll make changes to the original dataframe with using assigning operator.

# Creating column based on calculations:

For creating a column based on calculations, you have to write the following code:

df1['calculation']= df1["Age"] \* df1["Height"]

renaming a column

for renaming a column, we need to use rename() function. It will take a dictionary in column parameter and the key should be the old name and the value should be the new name. and if the parameter inplace is set to true so it will make changes to the original dataframe.

df1.rename(columns={"Team":"Country"}, inplace=True)

saving csv file:

you can save a df by using to\_csv() method but you have to set value of parameter index to False to avoid adding additional column to the df.

Handling date column:

If there is any date column in dataframe so it will be also a string format. So it is better to change it into datetime format. This will make the use of this column easy according to the requirements. For this purpose, we have to use a static function of panda’s library which will change that column to datetime format. Then you can do necessary operations easily on the column according to the requirements.

df["born\_year"] = pd.to\_datetime(dropdf["born\_date"], errors="coerce")

by setting errors parameter to coerce, this parameter will handle invalid values in the column and set it to Nan if they’re not convertible to datetime instead of raising an error.

# Handling null values

## To find how many nan values are there in all the columns:

To find this we’ll use 2 methods, df.isna() it’ll return the df with Boolean values with true and false. If we use sum() function at the end so it’ll sum up all the null values in the columns of df separately.

df.isna().sum()

filling null values

fo this purpose we can use the function fillna(). And if we give it a value so all nan columns will be filled with that value.

df["weight\_kg"].fillna(100)

but this is not a better option for filling. Sometimes its better to fill the values with mean. So for this purpose we can use this code:

df["weight\_kg"] = df["weight\_kg"].fillna(df["weight\_kg"].mean())

these nan can be filled with interpolate() method as well. And you can use the following code for this purpose:

df["height\_cm"].interpolate()

sometimes the rows containing nan values are dropped. For this purpose dropna() function is used like the following:

df.dropna()

but if you want to drop only those rows which has nan in a specific column so you can use the following code:

df1.dropna()

but then you need to run reset\_index() it will reassign index to the df because after dropping rows the index is out of order. If the parameter drop is set to true to it will delete the previous index and add new ordered index, if its default so it will keep the old unordered index as well in a column.

selecting top five or bottom five:

df[df["born\_country"]=="IND"]["born\_region"].value\_counts().head()

finding mean of athletes from a specific country:

df[df["born\_country"]=="JPN"]["height\_cm"].mean()

groupby():

The groupby() function in pandas is used to split data into groups based on a column or multiple columns. It allows performing aggregate, transformation, or filtration operations on these groups.

df.groupby(["born\_country"])["height\_cm"].mean().sort\_values(ascending=False).tail(10)

rank():

The rank() function in pandas assigns ranks to elements in a Series or DataFrame, with ties getting the average rank by default. It is useful for ordering data relative to other elements.

df[["name", "weight\_kg", "rank\_weight"]].sort\_values(by=["rank\_weight"])