Lesson8 Advanced Pandas Data Frames Last Update Feb 21, 2020

In the first Pandas Data Frame Lesson we were introduced to Pandas Data Frames. A data frame stores data as rows and columns similar to a spread sheet. Each column has a heading. Each row contains data entered into each column under the heading. In the previous lesson we had made a data frame to store information about cars having a make, model, number of doors, car type and price.

Make	Model	Doors	Price
Ford	Mustang	2	12000
GM	Spitfire	2	34000
Toyota	Yarris	4	26000
Nissan	Sentra	2	18000

To use a pandas data frame we must first import pandas into our python:

```
import pandas as pd
```

Next we load a data frame from the following csv file 'cars.csv'.

cars.csv

```
make,model,doors,price
Ford,Mustang,2,12000
GM,Spitfire,2,34000
Toyota,Yarris,4,26000
Nissan,Sentra,2,18000
```

We use the **read_csv** function to read the 'cars.csv' file into a data frame.

```
df = pd.read_csv('cars.csv')
```

We then print out the data frame using the pandas print statement and the data frame name df.

```
print(df)
```

```
     make     model  doors  price
0    Ford   Mustang      2  12000
1      GM  Spitfire      2  34000
2  Toyota    Yarris      4  26000
3  Nissan    Sentra      2   18000
```

Note: we have automatic supplied indexes 0 to 3 supplied to us. Automatic supplied Indexes represent a lookup index for each row. Row 0 is index 0.

Lookup a row by index

In this example we use the pandas **iloc** function to locate row index 1 and print row 1 out as a series. Where the : means all columns.

```
# print out row 1
print(df.iloc[1,:])
```

```
make           GM
model    Spitfire
doors           2
price       34000
Name: 1, dtype: object
```

Looking up a row by a index name

You can lookup a row by a index name instead of specifying a row number. First you need to set the index to a column name of your choice when you read in a csv file. You use the **index_col** parameter of the **read_csv** function to set a index to a specified column name . In the following example we read in a csv file and set the index column to column 'make'

```
df = pd.read_csv('cars.csv', index_col= 'make')
print(df)
```

```
           model  doors  price
make
Ford     Mustang      2  12000
GM      Spitfire      2  34000
Toyota    Yarris      4  26000
Nissan    Sentra      2   1800
```

We now have the ‘make’ column to be our row look up index. In this situation you can one of the index labels like ‘Ford’ to look up a row using the pandas **loc** function and returns a series.

```
# print out row 1 by label 'Ford' using index column 'make'
print(df.loc['Ford',:])
```

```
model    Mustang
doors          2
price      12000
Name: Ford, dtype: object
```

To continue this lesson we reload the data frame from the csv file.

```
df = pd.read_csv('cars.csv')
print(df)
```

```
     make     model  doors  price
0    Ford   Mustang      2  12000
1      GM  Spitfire      2  34000
2  Toyota    Yarris      4  26000
3  Nissan    Sentra      2   18000
```

Adding rows to a data frame

Next we want to add rows to a data frame using the function **append**.

To add rows to a data frame we first make a 1 row python dictionary with some data in each column of the row we want to add.

```
car = {'make':'BMW',
       'model':'Rover',
       'doors':2,
       'price':24000}
```

We then append the dictionary to the data frame as a new row of data.

df = df.append(car,ignore_index=True)

The **parameter ignore_index=** True means do not use the index labels when appending the row data, so the index labels continue the original sequence rather than starting at 0. The **append function** updates the original data frame.

The **append** function does not make a new data frame, it adds rows to the data frame in place. We now print out our updated data frame with our newly added row.

print(df)

```
     make     model  doors  price
0    Ford   Mustang      2  12000
1      GM  Spitfire      2  34000
2  Toyota    Yarris      4  26000
3  Nissan    Sentra      2   1800
4     BMW     Rover      2  24000
```

Adding a column to a data frame

To Add a new column to a data frame, we put the new column values in a list

mpg = [23,56,45,78,56]

We now have list of new column data to represent miles per gallon (mpg). Each value in the list will be a row value for the 'mpg' column.

To add new column to data frame you just assign the column name to list of column row list values.

df[new_column name'] = list of column values for new row

df['mpg'] = mpg
print(df)

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
2  Toyota    Yarris      4  26000   45
3  Nissan    Sentra      2   1800   78
4     BMW     Rover      2  24000   56
```

Alternately you can use the **assign** function**.** The **assign** function returns a new data frame, it does not update the original data frame.

df = df.assign(mpg = [23,56,45,78,56])
print(df)

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
2  Toyota    Yarris      4  26000   45
3  Nissan    Sentra      2   1800   78
4     BMW     Rover      2  24000   56
```

We use the head function to print first 5 rows of a data frame.

print(df.head())

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
2  Toyota    Yarris      4  26000   45
3  Nissan    Sentra      2   1800   78
4     BMW     Rover      2  24000   56
```

Saving a data frame to a file

You use the pandas **to_csv** function to save a data frame to a csv file.

df.to_csv(filename, index = False);

If you have a index column then use **index = false** argument

df.to_csv(filename, index = False);

Or else you will get a duplicate additional index column, called 'Unnamed'.

to do:

Save our cars data frame in the csv file named 'cars3.csv' that contains the 'mpg' column. We use index = false means do not print fields for index names.

df.to_csv('cars3.csv',index = False);

Data frame info

The panda **info()** method is used to get the complete information about the data set. It lists the column names, number of non null value counts and column data type's.

print(df.info())

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   make    5 non-null      object
 1   model   5 non-null      object
 2   doors   5 non-null      int64
 3   price   5 non-null      int64
 4   mpg     5 non-null      int64
dtypes: int64(3), object(2)
memory usage: 328.0+ bytes
```

Data frame statistics

The **describe** method computes summary statistics of number variables. The **describe** method returns statistical data about the data frame

df.describe()

```
          doors         price        mpg
count  5.000000      5.000000   5.000000
mean   2.400000  19560.000000  51.600000
std    0.894427  12671.542921  19.982492
min    2.000000   1800.000000  23.000000
25%    2.000000  12000.000000  45.000000
50%    2.000000  24000.000000  56.000000
75%    2.000000  26000.000000  56.000000
max    4.000000  34000.000000  78.000000
```

Pandas Statistical functions

Here are all the statistic functions you can apply to the entire data frame or to individual columns.

Function	Description
df.mean()	Returns the mean of all columns
df.corr()	Returns the correlation between columns in a data frame
df.count()	Returns the number of non-null values in each data frame column
df.max()	Returns the maximum value in each column the highest value in each column
df.min()	Returns the lowest value in each column the highest value in each column
df.median()	Returns the median of each column
df.std()	Returns the standard deviation of each column

Here is an example using mean on the entire data frame.

print(df.mean())

```
doors      2.4
price    19560.0
mpg        51.6
dtype: float64
```

Here is an example printing of mean of the ‘price’ column

print(df['price'].mean())

```
19560.0
```

to do:

Try out all the statistical functions’ for the whole data frame and then try out one of the columns of your choice.

Filter data

Filter uses the following boolean operators to select data by comparing row column values to a known value.

Operator	Description
>	greater than
<	less than
&	And (both)
|	Or (either)

Here is an example to filter cars by price greater than $20000. A new data frame is returned.

print(df[df['price']>20000])

```
    make     model  doors  price  mpg
1     GM  Spitfire      2  34000   56
2 Toyota    Yarris      4  26000   45
4    BMW     Rover      2  24000   56
```

The inner code **df['price']>20000])** evaluates the row 'price' column values as **True** or **False** depending if the row column value is greater than 2000. The resulting **True** and **False** values are then applied to the outer df then the results are printed to the screen.

```
0    False
1     True
2     True
3    False
4     True
```

Filtering two columns

We can also filter for price greater than 20000 and mpg > 40

print(df[(df['price']>20000) & (df['mpg']>40)])

We use 2 conditions **df['price']>20000** and **df['mpg']>40**

We use the & (and) operator to 'and' both conditions to test if both conditions are True.

(df['price']>20000) & (df['mpg']>40)])

We must enclose each condition in round brackets to force precedence. We want the conditions to be evaluated first then have the & (AND) operator applied to both conditions. '**And'** means **both** conditions must be true to have the result True. The result of the filter is as follows.

```
0    False
1     True
2     True
3    False
4     True
```

Three rows have been selected as True:

```
     make     model  doors  price  mpg
1      GM  Spitfire      2  34000   56
2  Toyota    Yarris      4  26000   45
4     BMW     Rover      2  24000   56
```

Sort columns

We use the **sort_values** function to sort a column ascending or descending. Ascending is the default.

To sort the 'mpg' column ascending:

print(df.sort_values('mpg'))

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
2  Toyota    Yarris      4  26000   45
1      GM  Spitfire      2  34000   56
4     BMW     Rover      2  24000   56
3  Nissan    Sentra      2   1800   78
```

to do: sort another column

The whole data fame is re-arranged accordingly to the sorted column values. Notice all the indexes have re-arranged accordingly to the mpg column. Sorting is done in place, if you just want to return a sorted data set then use **inplace=False**. In this situation a new data frame is returned and the original data frame does not change.

Example:

sorted_df = df.sort_values('mpg', inplace=False)

To sort the mpg column descending:

print(df.sort_values('mpg',ascending=False))

```
     make     model  doors  price  mpg
3  Nissan    Sentra      2   1800   78
1      GM  Spitfire      2  34000   56
4     BMW     Rover      2  24000   56
2  Toyota    Yarris      4  26000   45
0    Ford   Mustang      2  12000   23
```

Sorting multiple columns

We can sort on more than one column. In our case we want to sort by highest miles per gallon and lowest price.

Here we specify mpg as ascending and price as descending.

print(df.sort_values(by=['mpg','price'],ascending=[True,False]))

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
2  Toyota    Yarris      4  26000   45
4     BMW     Rover      2  24000   56
1      GM  Spitfire      2  34000   56
3  Nissan    Sentra      2  18000   78
```

Grouping a DataFrame by column values

A column may have many identical values that allows individual data frame's to be grouped together for each column value group, using the **groupby** function. A good example is to use the 'doors' column since we can group cars by 2 doors or 4 doors

g = df.groupby('doors')

The **groupby** function returns a **group** object that has functions to print out the **group values** as a **individual data frame** for each group value.

We use the **groups** function enclosed in a **list** to print out list of group values

print(list(g.groups))

```
[2, 4]
```

We can use the list of group values to print out the data frame for each group value in the group list using the **get_group** function.

```
#print out list of groups
for x in list(g.groups):
    print(g.get_group(x))
```

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
3  Nissan    Sentra      2   1800   78
4     BMW     Rover      2  24000   56

     make   model  doors  price  mpg
2  Toyota  Yarris      4  26000   45
```

Notice we have 2 different data frames. One for cars with 2 doors and one for cars with 4 doors.

We can print out one specific group data frame for a specified door.

print(g.get_group(2))

```
    make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
3  Nissan    Sentra      2   1800   78
4     BMW     Rover      2  24000   56
```

Alternatively we can set up a filter to print our cars with 2 doors

```
print(df[df['doors']==2])
```

```
  make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
3  Nissan    Sentra      2   1800   78
4     BMW     Rover      2  24000   56
```

Combinding groups to a new data frame using concat

The big problem is all the groups are in separate data frames, so we can combine all the groups to a new data frame in group order using the panda **concat** method, that joins together data frames.

```
# add groups to a empty data frame
df2 = pd.DataFrame()
for x in list(g.groups):
    df2 = pd.concat([df2,g.get_group(x)])
print(df2)
```

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
3  Nissan    Sentra      2  18000   78
4     BMW     Rover      2  24000   56
2  Toyota    Yarris      4  26000   45
```

Sorting each group by a column and concatenate to a new data frame.

Better still we can combine the groups together sorted by a column using the method **sort_values** and a column name to sort.

```
# add sorted groups to a empty data frame
df2 = pd.DataFrame()
for x in list(g.groups):
    df2 = pd.concat([df2,g.get_group(x).sort_values('mpg')])
print(df2)
```

```
     make     model  doors  price  mpg
0    Ford   Mustang      2  12000   23
1      GM  Spitfire      2  34000   56
4     BMW     Rover      2  24000   56
3  Nissan    Sentra      2  18000   78
2  Toyota    Yarris      4  26000   45
```

ADVANCED PANDAS HOMEWORK Question 1

Make a dictionary or a csv file with columns for animal types, names, age and sound they make.

Example: a cat the name is fluffy the age is 6 years old and the sound thy make is "purr"

Print out the data frame.

Group all animals together by type.

Print out each group details.

Make a new data frame that has the count of each animal and the average ages, and the sound it makes for each type.

Print out the data frame.

Make another data frame with columns animal type, name, the sound they make and animal type in order by groups where each group is orders by age.

Print out the data frame.

All data frames should have a title.

Save your py file as advpandas_homework1.py

You should get something like this:

```
animals:
   type      name sound  age
0   cat    fluffy  purr    5
1   dog     rover  bark    6
2  lion      tony  roar   11
3   cat  scratchy  purr    6
4   dog      spot  bark    4
5   dog    stanly  bark    3
6  lion    rudolf  roar    4



animal groups:

   type      name sound  age
0  cat    fluffy  purr    5
3  cat  scratchy  purr    6

   type    name sound  age
1  dog   rover  bark    6
4  dog    spot  bark    4
5  dog  stanly  bark    3

    type    name sound  age
2  lion    tony  roar   11
6  lion  rudolf  roar    4

animal types, counts and average ages:
   type count  average_age sound
0   cat     3     5.500000  purr
1   dog     3     4.333333  bark
2  lion     4     7.500000  roar

animal types sorted by age:
   type      name sound  age
6  lion    rudolf  roar    4
2  lion      tony  roar   11
5   dog    stanly  bark    3
4   dog      spot  bark    4
1   dog     rover  bark    6
0   cat    fluffy  purr    5
3   cat  scratchy  purr    6
```

pandas data types

Our known data types have corresponding pandas data types:
Strings are known as objects.

data type	Pandas data type	Example
float	float64, float32	1223.876
Int	int64, int32	543321
Datetime	datetime64[ns]	'2020-06-23
string	Object	'hello'

You may also specify the data type of the value to be replace NaN values with of the specified column.

df['column_name'].fillna(0).astype('int32')

The default value of 0 is float64 you may want int32 instead.

It is possible to change data type of a data frame using the **astype function**

df = df.astype(int)

There may be instance when you want to do this since a data frame has default data type **float** and you may want data type to be **int**.

The **to_numeric function** changes data frame to numeric by converting string types to numeric types. In this example we make a data frame with string numbers. The **to_numeric** function will convert the string numeric values to numeric values.

```
# make a data frame with column Numbers of string numeric values
df2 = pd.DataFrame({"Numbers" : ["1.234","23","-1234"]})
```

```
Numbers
0  1.234
1     23
2  -1234
```

We use the apply method to apply the **to_numeric** to each cell in the data frame

df2 = df2.apply(pd.to_numeric)
print(df2)

```
Numbers
0  1.234
1    23
2  -1234
```

Here we just change a specified data frame column to numeric

df['column_name'] = df['column_name'].apply(pd.to_numeric)

df2['Numbers'] = df2['Numbers'].apply(pd.to_numeric)

We can use **astype** function to change the data type of a specified column to a specified data type

df = df.astype({"column name": int})
df = df.astype({"Numbers": int})

printing out data types of all columns

We use the types property to print out data types of the whole data frame

print(df.dtypes)

```
make     object
model    object
doors     int64
price     int64
mpg       int32
dtype: object
```

Where **object** refers to a text string value

To continue this lesson we reload out data frame from csv file: cars2.csv containing NaN's. NaN means Not a Number which usually means a blank value in a csv file.

cars2.csv

```
make,model,doors,price,mpg
Ford,Mustang,2,12000,
GM,Spitfire,2,,56
Toyota,,4,26000,45
Nissan,Sentra,2,,78
BMW,Rover,2,24000,56
```

```
# reset data frame with nans
df = pd.read_csv('cars2.csv')
print(df)
```

```
     make     model  doors    price   mpg
0    Ford   Mustang      2  12000.0   NaN
1      GM  Spitfire      2      NaN  56.0
2  Toyota       NaN      4  26000.0  45.0
3  Nissan    Sentra      2      NaN  78.0
4     BMW     Rover      2  24000.0  56.0
```

Notice our data frame now has many NaN's placed in various locations.

Replacing null values in a data frame

We use the **fillna** function replace all Nan values in as data frame with 0. NaN's occur when loading a csv file into a data frame when some of the cells contain blank values.

```
df2  = df.fillna(0)
print(df2)
```

```
    make     model  doors    price   mpg
0    Ford   Mustang      2  12000.0   0.0
1      GM  Spitfire      2      0.0  56.0
2  Toyota         0      4  26000.0  45.0
3  Nissan    Sentra      2      0.0  78.0
4     BMW     Rover      2  24000.0  56.0
```

You can also replace null values in a specified column

df2 = df['column_name'].fillna(0)

df2 = df['mpg'].fillna(0)
print(df2)

```
0     0.0
1    56.0
2    45.0
3    78.0
4    56.0
Name: mpg, dtype: float64
```

Notice we return just the 'mpg' column as a series

You need to use **inplace=True** to update the original data frame.

df['column_name'].fillna(0,inplace=True)

df['mpg'].fillna(0,inplace=True)
print(df)

```
     make     model  doors    price   mpg
0    Ford   Mustang      2  12000.0   0.0
1      GM  Spitfire      2      NaN  56.0
2  Toyota       NaN      4  26000.0  45.0
3  Nissan    Sentra      2      NaN  78.0
4     BMW     Rover      2  24000.0  56.0
```

Notice we update the 'mpg' column in the whole data frame

You need to use **inplace=True** to update the original data frame.

df.fillna(0,inplace=True)
print(df)

The whole data frame is now updated with zeros.

```
    make     model  doors    price   mpg
0    Ford   Mustang      2  12000.0   0.0
1      GM  Spitfire      2      0.0  56.0
2  Toyota         0      4  26000.0  45.0
3  Nissan    Sentra      2      0.0  78.0
4     BMW     Rover      2  24000.0  56.0
```

Dropping rows in a data frame with NaN's

We now reload the data frame with NaNS

```
# reset data frame with nans
df = pd.read_csv('cars2.csv')
print(df)
```

```
     make     model  doors    price   mpg
0    Ford   Mustang      2  12000.0   NaN
1      GM  Spitfire      2      NaN  56.0
2  Toyota       NaN      4  26000.0  45.0
3  Nissan    Sentra      2      NaN  78.0
4     BMW     Rover      2  24000.0  56.0
```

Dropping all rows with NaN values

```
df2 = df.dropna()
print(df2)
```

```
  make  model  doors    price   mpg
4  BMW  Rover      2  24000.0  56.0
```

We are just left with the BMW row

Use **inplace=True** to update the original data frame.

```
df.dropna(inplace=True)
print(df)
```

```
  make  model  doors    price   mpg
4  BMW  Rover      2  24000.0  56.0
```

We are now just left with the BMW row

After you drop the rows with the NaN values you may want to reset the data frame indexes. Since many rows are gone

df.reset_index(drop=True, inplace=True)
print(df)

```
  make  model  doors    price   mpg
0  BMW  Rover      2  24000.0  56.0
```

We now reload the data frame with NaNS again

 # reset data frame with nans
df = pd.read_csv('cars2.csv')
print(df)

```
     make     model  doors    price   mpg
0    Ford   Mustang      2  12000.0   NaN
1      GM  Spitfire      2      NaN  56.0
2  Toyota       NaN      4  26000.0  45.0
3  Nissan    Sentra      2      NaN  78.0
4     BMW     Rover      2  24000.0  56.0
```

Dropping column's with NAN'S

You need to set tge axis to access columns. Use either one since axis =1 means axis ='columns'.

 # dropping columns with NaN's
df2 = df.dropna(axis='columns')
df2 = df.dropna(axis=1)
print(df2)

```
     make  doors
0    Ford      2
1      GM      2
2  Toyota      4
3  Nissan      2
4     BMW      2
```

Use **inplace=True** to update the original data frame.

df.dropna(axis='columns', inplace=True)
df.dropna(axis=1,inplace=True)
print(df)

```
     make  doors
0    Ford      2
1      GM      2
2  Toyota      4
3  Nissan      2
4     BMW      2
```

ADVANCED PANDAS HOMEWORK Question 2

Make a dataframe that has columns title, author, year published and price either from a dictionary, lists or csv file.

Fill the data frame with titles of your favourite book's.

If you used a csv file make some empty entries for the price and year columns.

If you are using a dictionary or a list use np.NaN to make some NaN entries in the year column.

Replace all NaN prices with the average price.

Use info() to print out the data types of each column

Change all the year columns to a string object

Use info() to print out the data types of each column

Print out the dataframe.

Save your py file as advpandas_homework2.py

You should get something like this:

```
data frame with NaN's
              title      author  year  price
0        wizard of oz   tom smith  1945   9.98
1           peter pan     joe toe  1867    NaN
2  alice in wonderland  bill jones  1867   5.78
3   three little bears  sam smelly  1978    NaN

data frame with NaN's replaced with average year
              title      author  year  price
0        wizard of oz   tom smith  1945   9.98
1           peter pan     joe toe  1867   7.88
2  alice in wonderland  bill jones  1867   5.78
3   three little bears  sam smelly  1978   7.88

data frame info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   title   4 non-null      object
 1   author  4 non-null      object
 2   year    4 non-null      int64
 3   price   4 non-null      float64
dtypes: float64(1), int64(1), object(2)
memory usage: 160.0+ bytes

data frame info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   title   4 non-null      object
 1   author  4 non-null      object
 2   year    4 non-null      object
 3   price   4 non-null      float64
dtypes: float64(1), object(3)
memory usage: 144.0+ bytes
```

END