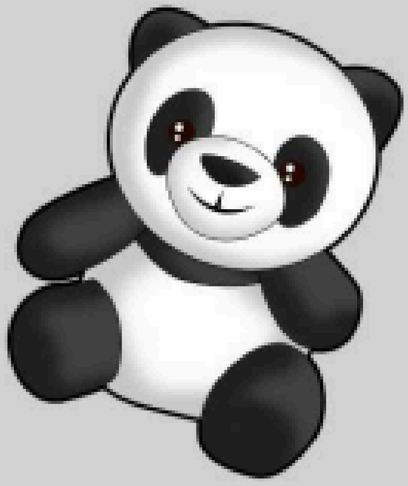
Data Analysis using Pandas Library

Python Pandas from Basics to Advance



Python Pandas

MANDAR PATIL

Data Analysis with Python Pandas

by Kaan Kabalak @ witfuldata.com

Pandas is the main Python library for data science and analytics. Whether you are building a machine learning model or just want to take a quick look at your data, you will use it. For this part, we are going to go over the main concepts of Pandas.

Let's begin

Loading and Inspecting the Data

First of all we need to import pandas. I also imported numpy because some of its functions can be very useful when we are analyzing data with pandas.

You can download the dataset from: https://www.kaggle.com/datasets/uciml/autompg-dataset

In [1]:
```
# Import
import numpy as np
import pandas as pd
```

We need to define a data frame with the read_csv function. This function takes a string file path as an argument. File path is basically the path where a file is located on your system. When I work with data, I usually put the data file in the same directory with the Python-Jupyter Notebook file, so that just passing the file name will be sufficient. Python requires only the name of a file if the file is in the same directory with the .py or .ipynb file you are running.

In [2]:
```
# Load the data from a csv file
auto_df = pd.read_csv("auto-mpg.csv")
```

In [3]:
```
# Check the first 6 rows
auto_df.head(6)
```

Out[3]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
5	15.0	8	429.0	198	4341	10.0	70	1	ford galaxie 500

This is a Pandas data frame. It has rows and columns. The rows have index numbers next to them (on the left). The row index starts from 0 (like most Python objects). The columns also have index numbers that start from 0 but they are visible to us like the row index numbers. We will learn how to do operations with these index numbers.

Let's take a look at several aspects of our data frame.

In [4]:

```
# Check the last 6 rows
auto_df.tail(6)
```

Out[4]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
392	27.0	4	151.0	90	2950	17.3	82	1	chevrolet camaro
393	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
396	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
397	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

In [5]:

```
# Check the column names
auto_df.columns
```

Out[5]:

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'model year', 'origin', 'car name'],
      dtype='object')
```

In [6]:
```
# You can view column names as a list
list(auto_df.columns)
```

Out[6]:
```
['mpg',
 'cylinders',
 'displacement',
 'horsepower',
 'weight',
 'acceleration',
 'model year',
 'origin',
 'car name']
```

In [7]:
```
# Check the shape (number of rows, number of columns)
auto_df.shape
```

Out[7]:
```
(398, 9)
```

In [8]:
```
# Check the size attribute (number of rows x number of columns)
auto_df.size
```

Out[8]:
```
3582
```

We can use the .info ( ) method to learn some very important things about our data frame such as:

RangeIndex : The number of entries (rows) ad the range of their index numbers. '#' : The index number of columns Column : Column name Non-Null Count: The number of non-null values. Dtype : The data type of values that are held by the column. (object usually stands for string)

In [9]:
```
# Check the info
auto_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    398 non-null    object 
 4   weight        398 non-null    int64  
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64  
 7   origin        398 non-null    int64  
 8   car name      398 non-null    object 
dtypes: float64(3), int64(4), object(2)
memory usage: 28.1+ KB
```

Something seems odd here, doesn't it?

The horsepower should hold numerical (integer or float) values but it says here that it has object (string) data type. Let's see how we can find out why and solve this issue.

We can call the unique on the column name to check all unique values.

In [10]:
```
auto_df.horsepower.unique()
```

Out[10]:
```
array(['130', '165', '150', '140', '198', '220', '215', '225', '190',
       '170', '160', '95', '97', '85', '88', '46', '87', '90', '113',
       '200', '210', '193', '?', '100', '105', '175', '153', '180', '110',
       '72', '86', '70', '76', '65', '69', '60', '80', '54', '208', '155',
       '112', '92', '145', '137', '158', '167', '94', '107', '230', '49',
       '75', '91', '122', '67', '83', '78', '52', '61', '93', '148',
       '129', '96', '71', '98', '115', '53', '81', '79', '120', '152',
       '102', '108', '68', '58', '149', '89', '63', '48', '66', '139',
       '103', '125', '133', '138', '135', '142', '77', '62', '132', '84',
       '64', '74', '116', '82'], dtype=object)
```

Looking at these values carefully, we can see that there are entries marked with '?'. Marking an entry like this may cause the data type of the whole column to change to string because a Pandas data frame column can only hold one type of data. If there is a string value, then all other values under the same column must be a string.

I will get into the details on how to solve such problems in the future. For now, let's take a look at what we can do to easily fix this. We will use the na_values= parameter of the read_csv function to turn '?' values into NaN (missing) values. NaN values are treated as unidentified floats by Pandas. This will turn our column to a numeric (float, in this case) data type. This will allow us to carry out arithmetic operations easily.

In [11]:
```python
# Pass na_values like a keyword argument and set its value to '?'
auto_df = pd.read_csv('auto-mpg.csv', na_values='?')

# Call the function method again
auto_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    392 non-null    float64
 4   weight        398 non-null    int64  
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64  
 7   origin        398 non-null    int64  
 8   car name      398 non-null    object 
dtypes: float64(4), int64(4), object(1)
memory usage: 28.1+ KB
```

Here, now the horsepower column holds numerical float data as it should.

Selecting Data

For this section, we will be working with the concrete dataset.

You can download it from:

https://www.kaggle.com/datasets/prathamtripathi/regression-with-neural-networking

It is beneficial to take a look at the data dictionary before starting. Data dictionaries are documents that explain the meaning of variables in the dataset:

Compressive strength data:

"Cement" - Portland cement in kg/m3

"Blast Furnace Slag" - Blast furnace slag in kg/m3

"Fly Ash" - Fly ash in kg/m3

"Water" - Water in liters/m3

"Superplasticizer" - Superplasticizer additive in kg/m3

"Coarse Aggregate" - Coarse aggregate (gravel) in kg/m3

"Fine Aggregate" - Fine aggregate (sand) in kg/m3

"Age" - Age of the sample in days

"Strength" - Concrete compressive strength in megapascals (MPa)

In [12]:
```
# Load the data
concrete_df = pd.read_csv("concrete_data.csv")
concrete_df.head(5)
```

Out[12]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

Slicing with Column Names

We can use slicing with brackets [ ] and the name of the column(s) to see the data.

In [13]:
```
# Values of a single column
concrete_df['Cement']
```

Out[13]:
```
0       540.0
1       540.0
2       332.5
3       332.5
4       198.6
        ...
1025    276.4
1026    322.2
1027    148.5
1028    159.1
1029    260.9
Name: Cement, Length: 1030, dtype: float64
```

This shows us the row index numbers and the values. If we want to see the result like a data frame with column names, we can use double brakcets like this:

In [14]:
```
# Values of a single column as a data frame
concrete_df[['Cement']]
```

Out[14]:

	Cement
0	540.0
1	540.0
2	332.5
3	332.5
4	198.6
...	...
1025	276.4
1026	322.2
1027	148.5
1028	159.1
1029	260.9

1030 rows × 1 columns

In [15]:
```
# Data frame of 2 (or more) columns
concrete_df[['Cement','Age']]
```

Out[15]:

	Cement	Age
0	540.0	28
1	540.0	28
2	332.5	270
3	332.5	365
4	198.6	360
...	...	...
1025	276.4	28
1026	322.2	28
1027	148.5	28
1028	159.1	28
1029	260.9	28

1030 rows × 2 columns

In [16]:
```
# Data frame of multiple columns
concrete_df[['Blast Furnace Slag','Water','Strength']]
```

Out[16]:

	Blast Furnace Slag	Water	Strength
0	0.0	162.0	79.99
1	0.0	162.0	61.89
2	142.5	228.0	40.27
3	142.5	228.0	41.05
4	132.4	192.0	44.30
...	...	...	...
1025	116.0	179.6	44.28
1026	0.0	196.0	31.18
1027	139.4	192.7	23.70
1028	186.7	175.6	32.77
1029	100.5	200.6	32.40

1030 rows × 3 columns

Using .loc & .iloc

We can also use .loc and .iloc to get subsets of the data.

- .loc works with row index numbers and column names.
- .iloc works only with row and column index numbers.

Let's take a look at how they work:

In [17]:
```
# Using .loc to access the values of single column
concrete_df.loc[:,'Strength'] # the : means "select all rows"
```

Out[17]:
```
0       79.99
1       61.89
2       40.27
3       41.05
4       44.30
        ...
1025    44.28
1026    31.18
1027    23.70
1028    32.77
1029    32.40
Name: Strength, Length: 1030, dtype: float64
```

In [18]:
```
# Using .loc to access the values of a single column like a data frame
concrete_df.loc[:,['Strength']]
```

Out[18]:

	Strength
0	79.99
1	61.89
2	40.27
3	41.05
4	44.30
...	...
1025	44.28
1026	31.18
1027	23.70
1028	32.77
1029	32.40

1030 rows × 1 columns

In [19]:
```
# Using .loc for a data frame from multiple columns
concrete_df.loc[:,['Cement','Water','Strength']]
```

Out[19]:

	Cement	Water	Strength
0	540.0	162.0	79.99
1	540.0	162.0	61.89
2	332.5	228.0	40.27
3	332.5	228.0	41.05
4	198.6	192.0	44.30
...	...	...	...
1025	276.4	179.6	44.28
1026	322.2	196.0	31.18
1027	148.5	192.7	23.70
1028	159.1	175.6	32.77
1029	260.9	200.6	32.40

1030 rows × 3 columns

We do not have to select all rows or columns with .loc. We can specify a range for them. See the examples below:

In [20]:
```
# Select rows from 0 to 200, select columns from Cement to Fine Aggreagate
concrete_df.loc [0:200,"Cement":"Fine Aggregate"]
```

Out[20]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5
...	...	...	...	...	...	...	...
196	194.7	0.0	100.5	165.6	7.5	1006.4	905.9
197	194.7	0.0	100.5	165.6	7.5	1006.4	905.9
198	194.7	0.0	100.5	165.6	7.5	1006.4	905.9
199	190.7	0.0	125.4	162.1	7.8	1090.0	804.0
200	190.7	0.0	125.4	162.1	7.8	1090.0	804.0

201 rows × 7 columns

We use .iloc when we want to use only the index numbers for rows and columns. Just like with .loc, we can specify a range.

In [21]:

```
# Using .iloc with index numbers (Select the first 100 rows, select the first colum
concrete_df.iloc[0:100,[0]]
```

Out[21]:

	Cement
0	540.0
1	540.0
2	332.5
3	332.5
4	198.6
...	...
95	425.0
96	425.0
97	375.0
98	475.0
99	469.0

100 rows × 1 columns

In [22]:

```
# Using .iloc with index number (This time with multiple columns)
concrete_df.iloc[0:100,[0,2,4]]
```

Out[22]:

	Cement	Fly Ash	Superplasticizer
0	540.0	0.0	2.5
1	540.0	0.0	2.5
2	332.5	0.0	0.0
3	332.5	0.0	0.0
4	198.6	0.0	0.0
...	...	...	...
95	425.0	0.0	16.5
96	425.0	0.0	18.6
97	375.0	0.0	23.4
98	475.0	0.0	8.9
99	469.0	0.0	32.2

100 rows × 3 columns

Note: .loc and .iloc behave a bit differently with ranges. .loc ranges are all inlusive while .iloc ranges are inclusive before : and exclusive after :

This means that

- .loc[0:50, ['cement']] > will give you rows with index numbers from 0 to 50 (50 included) --- A total of 51 rows
- .iloc[0:50, [0]] > will give you rows with index numbers from 0 **up to** 50 (50 excluded) --- A total of 50 rows

In [23]:
```
# Rows from 0 to 150, columns from 0 to 5
concrete_df.iloc[0:150,0:5]
```

Out[23]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer
0	540.0	0.0	0.0	162.0	2.5
1	540.0	0.0	0.0	162.0	2.5
2	332.5	142.5	0.0	228.0	0.0
3	332.5	142.5	0.0	228.0	0.0
4	198.6	132.4	0.0	192.0	0.0
...	...	...	...	...	...
145	469.0	117.2	0.0	137.8	32.2
146	425.0	106.3	0.0	153.5	16.5
147	388.6	97.1	0.0	157.9	12.1
148	531.3	0.0	0.0	141.8	28.2
149	425.0	106.3	0.0	153.5	16.5

150 rows × 5 columns

Note: If you use the .iloc without specifying column names after a comma, it will select all columns

In [24]:
```
# Using .iloc without specifying columns
concrete_df.iloc[0:15]
```

Out[24]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30
5	266.0	114.0	0.0	228.0	0.0	932.0	670.0	90	47.03
6	380.0	95.0	0.0	228.0	0.0	932.0	594.0	365	43.70
7	380.0	95.0	0.0	228.0	0.0	932.0	594.0	28	36.45
8	266.0	114.0	0.0	228.0	0.0	932.0	670.0	28	45.85
9	475.0	0.0	0.0	228.0	0.0	932.0	594.0	28	39.29
10	198.6	132.4	0.0	192.0	0.0	978.4	825.5	90	38.07
11	198.6	132.4	0.0	192.0	0.0	978.4	825.5	28	28.02
12	427.5	47.5	0.0	228.0	0.0	932.0	594.0	270	43.01
13	190.0	190.0	0.0	228.0	0.0	932.0	670.0	90	42.33
14	304.0	76.0	0.0	228.0	0.0	932.0	670.0	28	47.81

Sorting Values

Pandas .sort_values method allows us to sort values by a column in a certain order.

For this section we will use the automobile data we used in the first section.

Let's see some examples:

In [25]:
```
# Sort the values by a column, in descending order (ascending = False), ignore the
sorted_auto = auto_df.sort_values(by='mpg', ascending=False, ignore_index=True)
sorted_auto.head(5)
```

Out[25]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	46.6	4	86.0	65.0	2110	17.9	80	3	mazda glc
1	44.6	4	91.0	67.0	1850	13.8	80	3	honda civic 1500 gl
2	44.3	4	90.0	48.0	2085	21.7	80	2	vw rabbit c (diesel)
3	44.0	4	97.0	52.0	2130	24.6	82	2	vw pickup
4	43.4	4	90.0	48.0	2335	23.7	80	2	vw dasher (diesel)

If we set ignore_index to False, the original row index numbers will appear.

In [26]:
```
# Ignore_index
sorted_auto_orinx = auto_df.sort_values(by='mpg', ascending=False, ignore_index=Fal
sorted_auto_orinx.head(5)
```

Out[26]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
322	46.6	4	86.0	65.0	2110	17.9	80	3	mazda glc
329	44.6	4	91.0	67.0	1850	13.8	80	3	honda civic 1500 gl
325	44.3	4	90.0	48.0	2085	21.7	80	2	vw rabbit c (diesel)
394	44.0	4	97.0	52.0	2130	24.6	82	2	vw pickup
326	43.4	4	90.0	48.0	2335	23.7	80	2	vw dasher (diesel)

Let's see some different examples:

In [27]:
```
# Sorted example
sorted_weight = auto_df.sort_values(by='weight', ascending=False, ignore_index=True
sorted_weight.head(5)
```

Out[27]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	13.0	8	400.0	175.0	5140	12.0	71	1	pontiac safari (sw)
1	11.0	8	400.0	150.0	4997	14.0	73	1	chevrolet impala
2	12.0	8	383.0	180.0	4955	11.5	71	1	dodge monaco (sw)
3	12.0	8	429.0	198.0	4952	11.5	73	1	mercury marquis brougham
4	12.0	8	455.0	225.0	4951	11.0	73	1	buick electra 225 custom

In [28]:

```
# Sorted example in ascending order
sorted_displ = auto_df.sort_values(by='displacement', ascending=True, ignore_index=
sorted_displ.head(5)
```

Out[28]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	29.0	4	68.0	49.0	1867	19.5	73	2	fiat 128
1	19.0	3	70.0	97.0	2330	13.5	72	3	mazda rx2 coupe
2	18.0	3	70.0	90.0	2124	13.5	73	3	maxda rx3
3	23.7	3	70.0	100.0	2420	12.5	80	3	mazda rx-7 gs
4	32.0	4	71.0	65.0	1836	21.0	74	3	toyota corolla 1200

Descriptive Summary Statistics

We can use the .describe( ) method to see summary descriptive statistics about the columns of our data frame:

In [29]:

```
# Check summary statistics
auto_df.describe()
```

Out[29]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year
count	398.000000	398.000000	398.000000	392.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.469388	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	38.491160	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	93.500000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	126.000000	3608.000000	17.175000	79.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000

So, what does this tell us? Let's take a look at the horsepower column to understand better.

- The count tells us that the information on horsepower has been collected from 392 cars. There are 398 observations.
- The mean tells us that the average horsepower for the cars is 104
- The std (standart deviation) shows us the variety of horsepower values. A car, by average, has 38 more or less than the mean of all horsepower values
- The min stands for the minimum value
- %25 stands for the first percentile. What does it mean? It means that that the car which has more horsepower than %25 of the cars has 75 horsepower
- %50 stands for the second percentile or the median. It represents the car which has more horsepower than %50 of the cars.
- %75 stands for the third percentile. Just like the first and the second ones, it represents the car which has more horsepower than %75 of cars.
- The max stands for the maximum value

Filtering

We can form filters with operators like ==, !=, <,>,>=,<=,& (AND), | (OR). What these operators do is explained in the part about control flow statements.

There are two main approaches we can use. The first one (and my favorite) is to form a filter and assign it to a variable. Then, we can use this filter variable to get a subset of the data frame through slicing. See the example below:

In [30]:
```
# Form a filter and assign it to a variable
filter_one = concrete_df['Age'] > 100
```

In [31]:
```
concrete_df[filter_one]
```

Out[31]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30
6	380.0	95.0	0.0	228.0	0.0	932.0	594.0	365	43.70
12	427.5	47.5	0.0	228.0	0.0	932.0	594.0	270	43.01
...	...	...	...	...	...	...	...	...	...
798	500.0	0.0	0.0	200.0	0.0	1125.0	613.0	270	55.16
813	310.0	0.0	0.0	192.0	0.0	970.0	850.0	180	37.33
814	310.0	0.0	0.0	192.0	0.0	970.0	850.0	360	38.11
820	525.0	0.0	0.0	189.0	0.0	1125.0	613.0	270	67.11
823	322.0	0.0	0.0	203.0	0.0	974.0	800.0	180	29.59

62 rows × 9 columns

In [32]:
```
# Form a filter with multiple conditions
filter_two = (concrete_df['Age']>120) & (concrete_df['Cement']>380)
concrete_df[filter_two]
```

Out[32]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
12	427.5	47.5	0.0	228.0	0.0	932.0	594.0	270	43.01
19	475.0	0.0	0.0	228.0	0.0	932.0	594.0	180	42.62
20	427.5	47.5	0.0	228.0	0.0	932.0	594.0	180	41.84
33	475.0	0.0	0.0	228.0	0.0	932.0	594.0	270	42.13
41	427.5	47.5	0.0	228.0	0.0	932.0	594.0	365	43.70
56	475.0	0.0	0.0	228.0	0.0	932.0	594.0	365	41.93
755	540.0	0.0	0.0	173.0	0.0	1125.0	613.0	180	71.62
756	540.0	0.0	0.0	173.0	0.0	1125.0	613.0	270	74.17
795	525.0	0.0	0.0	189.0	0.0	1125.0	613.0	180	61.92
797	500.0	0.0	0.0	200.0	0.0	1125.0	613.0	180	51.04
798	500.0	0.0	0.0	200.0	0.0	1125.0	613.0	270	55.16
820	525.0	0.0	0.0	189.0	0.0	1125.0	613.0	270	67.11

The second approach is to write filters without assigning them to variables. I don't recommend doing this because they can look too cluttered. Also, the first approach is much more reproducible.

In [33]:
```
# Filtering without variable assignment
concrete_df[(concrete_df['Age']>120) & (concrete_df['Blast Furnace Slag']>=140)]
```

Out[33]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
23	139.6	209.4	0.0	192.0	0.0	1047.0	806.9	180	44.21
34	190.0	190.0	0.0	228.0	0.0	932.0	670.0	365	53.69
35	237.5	237.5	0.0	228.0	0.0	932.0	594.0	270	38.41
39	237.5	237.5	0.0	228.0	0.0	932.0	594.0	180	36.25
42	237.5	237.5	0.0	228.0	0.0	932.0	594.0	365	39.00
50	332.5	142.5	0.0	228.0	0.0	932.0	594.0	180	39.78
51	190.0	190.0	0.0	228.0	0.0	932.0	670.0	180	46.93
63	190.0	190.0	0.0	228.0	0.0	932.0	670.0	270	50.66
66	139.6	209.4	0.0	192.0	0.0	1047.0	806.9	360	44.70

Grouping & Aggregation

Let's understand what grouping and aggregation are:

- Grouping --- Forming groups from a column's values. For example, we have the 'origin' column in the automobile dataset. Every row of data tells us if the origin of the car is 1 (USA), 2 (Europe) or 3 (Asia). There are 398 rows in the dataset, meaning that there are 398 row values under the 'origin' column with a value representing one of these 3 origins. Here, we can form a group based on the origin of the automobiles. Instead of considering them through individual row values, we can get an overview of all automobiles organized into these 3 groups.

- Aggregation --- After grouping our data, we can look at values of different columns based on the groups we have. For example, we can look at the values of the 'weight' column according to each group. To make things more insightful, we can use an aggregate function on the column values we have. In the example below, we look at the average weight based on the origin groups by using the mean aggregate function

Note: In mathematical computation, an aggregate function is a function that takes multiple values as an input to produce a single output. Some of the most used aggregate functions are:

- Sum
- Count
- Min
- Max
- Mean
- Median

In [34]:

```
# Group by a column
grouped_origin = auto_df.groupby('origin')
```

In [35]:
```
# The mean of the weight column for each origin group
grouped_origin['weight'].mean()
```

Out[35]:
```
origin
1    3361.931727
2    2423.300000
3    2221.227848
Name: weight, dtype: float64
```

In [36]:
```
# The max of mpg for each origin group
grouped_origin['mpg'].max()
```

Out[36]:
```
origin
1    39.0
2    44.3
3    46.6
Name: mpg, dtype: float64
```

In [37]:
```
# Max accelaration for each cylinder number group
grouped_cylinder = auto_df.groupby('cylinders')
grouped_cylinder['acceleration'].max()
```

Out[37]:
```
cylinders
3    13.5
4    24.8
5    20.1
6    21.0
8    22.2
Name: acceleration, dtype: float64
```

In [38]:
```
# Standart deviation of mpg (miles-per-gallon) for each cylinder number group
grouped_cylinder['mpg'].std()
```

Out[38]:
```
cylinders
3    2.564501
4    5.710156
5    8.228204
6    3.807322
8    2.836284
Name: mpg, dtype: float64
```

In [39]:
```
# Such groupby object aggregation results can also be accessed like dataframes by u
grouped_cylinder[['mpg']].std()
```

Out[39]:

	mpg
cylinders	
3	2.564501
4	5.710156
5	8.228204
6	3.807322
8	2.836284

In [40]:
```
# The mean aggregated results like a data frame
grouped_cylinder[['mpg']].mean()
```

Out[40]:

	mpg
cylinders	
3	20.550000
4	29.286765
5	27.366667
6	19.985714
8	14.963107

In [41]:
```
# Group the concrete dataset based on age
grouped_concrete = concrete_df.groupby('Age')
```

In [42]:
```
# Median strength for each age group
grouped_concrete[['Strength']].median()
```

Out[42]:

	Strength
Age	
1	9.455
3	15.720
7	21.650
14	26.540
28	33.760
56	51.720
90	39.680
91	67.950
100	46.985
120	39.380
180	40.905
270	51.730
360	41.685
365	42.815

Adding New Columns

Before we finish, it would be nice to take a look at how we can add new columns.

The main rule we have to take into consideration here is that the column we are to add has to have the same length (number of rows) as the rest of the data frame.

We can decide to manually fill in the column values or we can use methods and functions to fill in the new column with processed or aggregated values. You will most likely go with the second approach as it is more practical, faster and easier.

For our example, we will add a new column to the concrete dataset, which will show the strength/cement ratio.

In [43]:
```
concrete_df['RatioStrCem'] = concrete_df['Strength'] / concrete_df['Cement']
concrete_df
```

Out[43]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength	Ra
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99	
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89	
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27	
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05	
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30	
...	...	...	...	...	...	...	...	...	...	
1025	276.4	116.0	90.3	179.6	8.9	870.1	768.3	28	44.28	
1026	322.2	0.0	115.6	196.0	10.4	817.9	813.4	28	31.18	
1027	148.5	139.4	108.6	192.7	6.1	892.4	780.0	28	23.70	
1028	159.1	186.7	0.0	175.6	11.3	989.6	788.9	28	32.77	
1029	260.9	100.5	78.3	200.6	8.6	864.5	761.5	28	32.40	

1030 rows × 10 columns

Exercises

You can find the diabetes data here: https://www.kaggle.com/datasets/akshaydattatraykhare/diabetes-dataset

- Check the first 7 rows of the diabetes data.
- How many rows does the diabetes data have?
- What are the column names of the diabetes data?
- What is the size of the diabetes data?
- What is the shape of the concrete data?
- Check the last 3 rows of the concrete data.
- Select the first 30 rows of the second, fourth and the fifth columns of the concrete dataset
- Form a data frame from the mpg, cylinders and the displacement columns of the auto-mpg dataset
- Sort the concrete data by strength in ASCENDING order, select the first 20 rows of strength and cement columns

- Sort the concrete data by age in DESCENDING order, select the first 15 rows of age and strength columns
- Sort the diabetes data by glucose in DESCENDING order, select the first 12 rows of glucose and bmi columns
- Sort the auto dataset by accelaration in ASCENDING order, select the first 15 rows of accelaration, mpg, displacement and weight columns
- Patients with a glucose higher than 120 AND blood pressure higher than or equal to 68 (diabetes data)
- Patients with glucose higher than 140 AND bmi lower than 27 (diabetes data)
- Concrete with water higher than 200 AND age lower than 300
- Automobiles with mpg rate higher than 15 AND cylinder number higher than 6
- Automobiles with mpg higher than 20 AND weight lower than 4000 AND acceleration higher than or equal to 15
- Automobiles with mpg higher than 25 OR acceleration higher than or equal to 20
- What is the average glucose level based on diabetes outcome?
- Access the minimum strength values for each age group of the concrete dataset.
- Access the maximum bmi values based on diabetes outcome.
- Access the standart deviation of weight for each origin group of the auto dataset.