Data Pre-processing

You should now be able to import datasets from various sources and explore the look and feel of the data. Handling missing values, creating dummy variables and plots are some tasks that an analyst (predictive modeller) does with almost all the datasets to make them model-worthy.

At the end of this lab you should be comfortable with the following functions:

- **Sub-set a dataset**: Slicing and dicing data, selecting few rows and columns based on certain conditions that is similar to filtering in Excel
- **Generating random numbers**: Generating random numbers is an important tool while performing simulations and creating dummy data frames
- **Aggregating data**: A technique that helps to group the data by categories in the categorical variable
- **Sampling data**: This is very important before venturing into the actual modelling; dividing a dataset between training and testing data is essential
- **Merging and appending/concatenating datasets**: This is the solution of the problem that arises when the data required for the purpose of modelling is scattered over different datasets

We will be using a variety of public datasets in this lab. Another good way of demonstrating these concepts is to use dummy datasets created using random numbers. In fact, random numbers are used heavily for this purpose. We will be using a mix of both public datasets and dummy datasets, created using random numbers.

Subsetting a dataset

The task of subsetting a dataset can entail a lot of things. Let us look at them one by one. In order to demonstrate it, let us first import the `Customer Churn Model` dataset, which we used in the last lab:

```
import pandas as pd
data=pd.read_csv('c:/FILE_PATH/Customer Churn Model.txt')
```

Selecting columns

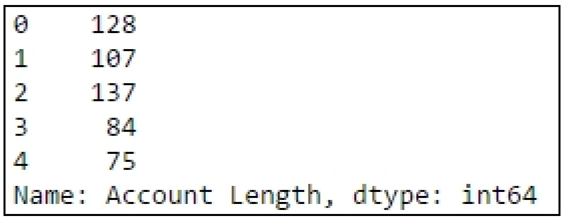
Very frequently, an analyst might come across situations wherein only a handful of columns among a vast number of columns are useful and are required in the model. It then becomes important, to select particular columns. Let us see how to do that.

If one wishes to select the `Account Length` variable of the data frame we just imported, one can simply write:

```
account_length=data['Account Length']
account_length.head()
```

The square bracket (`[ ]`) syntax is used to subset a column of a data frame. One just needs to type the appropriate column name in the square brackets. Selecting one column returns a

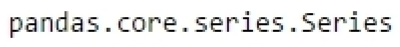
`Series` object (an object similar to a data frame) consisting of the values of the selected column. The output of the preceding snippet is as follows:



```
0    128
1    107
2    137
3     84
4     75
Name: Account Length, dtype: int64
```

First few entries of the Account Length column

The fact that this process returns a series can be confirmed by typing `type(account_length)`, this will return something similar to the following output, as a result:



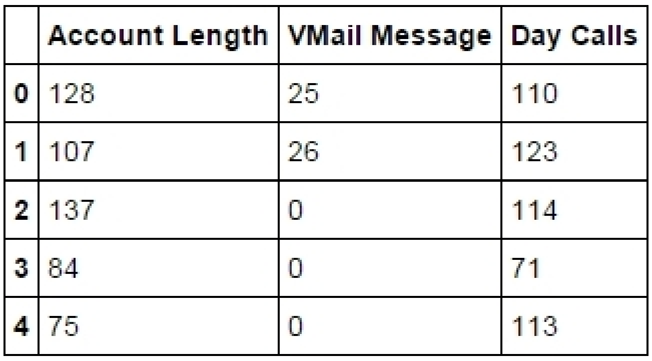
```
pandas.core.series.Series
```

Selecting multiple columns can be accomplished in a similar fashion. One just needs to add an extra square bracket to indicate that it is a list of column names that they are selecting and not just one column.

If one wants to select `Account Length`, `VMail Message`, and `Day Calls`, one can write the code, as follows:

```
Subdata = data[['Account Length','VMail Message','Day Calls']]
subdata.head()
```

The output of the preceding snippet should be similar to the following screenshot:

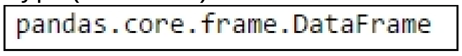


	Account Length	VMail Message	Day Calls
0	128	25	110
1	107	26	123
2	137	0	114
3	84	0	71
4	75	0	113

First few entries of the Account Length and VMail Message columns

Unlike in the case of selecting a single column, selecting multiple columns throws up a data frame, as the result:

```
type(subdata)
```



```
pandas.core.frame.DataFrame
```

One can also create a list of required columns and pass the list name as the parameter inside the square bracket to subset a data frame. The following code snippet will give the same result, as shown in the next section:

```
wanted_columns=['Account Length','VMail Message','Day Calls']
subdata=data[wanted]
subdata.head()
```

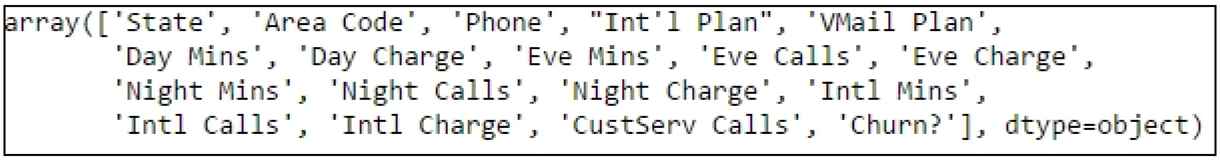
In some cases, one might want to delete or remove certain columns from the dataset before they proceed to modelling. The same approach, as taken in the preceding section, can be taken in such cases.

This approach of subsetting columns from data frames works fine when the list of columns is relatively small (3-5 columns). After this, the time consumed in typing column names warrants some more efficient methods to do this. The trick is to manually create a list to complement (a list not containing the elements that are present in the other set) the bigger list and create the bigger list using looping. The complement list of a big table will always be small; hence, we need to make the method a tad bit efficient.

Let us have a look at the following code snippet to observe how to implement this:

```
wanted=['Account Length','VMail Message','Day Calls']
column_list=data.columns.values.tolist()
sublist=[x for x in column_list if x not in wanted]
subdata=data[sublist]
subdata.head()
```

The `sublist` as expected contains all the column names except the ones listed in the `wanted` list, as shown in the following screenshot:



```
array(['State', 'Area Code', 'Phone', "Int'l Plan", 'VMail Plan',
       'Day Mins', 'Day Charge', 'Eve Mins', 'Eve Calls', 'Eve Charge',
       'Night Mins', 'Night Calls', 'Night Charge', 'Intl Mins',
       'Intl Calls', 'Intl Charge', 'CustServ Calls', 'Churn?'], dtype=object)
```

Column names of the subdata data frame

In the third line of the preceding code snippet, a list comprehension has been used. It is a convenient method to run for loops over lists and get lists as output. Many of you, who have experience with Python, will know of this. For others, it is not rocket science; just a better way to run `for` loops.

Selecting rows

Selecting rows is similar to selecting columns, in the sense that the same square bracket is used, but instead of column names the row number or indices are used. Let us see some examples to know how to select a particular number of rows from a data frame:

- If one wants to select the first `50` rows of the data frame, one can just write:

  ```
  data[1:50]
  ```

- It is important to note that one needs to pass a range of numbers to subset a data frame over rows. To select 50 rows starting from 25th column, we will write:

  ```
  data[25:75]
  ```

- If the lower limit is not mentioned, it denotes that the upper limit is the starting row of the data, which is row 1 in most cases. Thus, `data[:50]` is similar to `data[1:50]`.

In the same way, if the upper limit is not mentioned, it is assumed to be the last row of the dataset. To select all the rows except the first 50 rows, we will write `data[51:]`.

A variety of permutations and combinations can be performed on these rules to fetch the row that one needs.

Another important way to subset a data frame by rows is conditional or Boolean subsetting. In this method, one filters the rows that satisfy certain conditions. The condition can be either an inequality or a comparison written inside the square bracket. Let us see a few examples of how one can go about implementing them:

- Suppose, one wants to filter the rows that have clocked `Total Mins` to be greater than 500. This can be done as follows:

  ```
  data1=data[data['Total Mins']>500]
  data1.shape
  ```
- The newly created data frame, after filtering, has 2720 rows compared to 3333 in the unfiltered data frame. Clearly, the balance rows have been filtered by the condition.

- Let us have a look at another example, where we provide equality as a condition. Let us filter the rows for which the state is `VA`:

  ```
  data1=data[data['State']=='VA']
  data1.shape
  ```

- This data frame contains only 77 rows, while the rest get filtered.

- One can combine multiple conditions, as well, using `AND` (`&`) and `OR` (`|`) operators. To filter all the rows in the state `VA` that have `Total Mins` greater than 500, we can write:

  ```
  data1=data[(data['Total Mins']>500) & (data['State']=='VA')]
  data1.shape
  ```

- This data frame contains only 64 rows; it's lesser than the previous data frame. It also has two conditions, both of which must be satisfied to get filtered. The `AND` operator has a subtractive effect.

- To filter all the rows that are either in state `VA` or have `Total Mins` greater than 500, we can write the following code:

  ```
  data1=data[(data['Total Mins']>500) | (data['State']=='VA')]
  data1.shape
  ```

- This data frame has 2733 rows, which is greater than 2720 rows obtained with just one filter of `Total Mins` being greater than 500. The `OR` operator has an additive affect.

Selecting a combination of rows and columns

This is the most used form of subsetting a dataset. Earlier in this lab we selected three columns of this dataset and called the sub-setted data frame a `subdata`. What if we wish to look at specific rows of that sub-setted data frame? How can we do that? We just need another square bracket adjacent to the one already existing.

Let's say, we need to look at the first 50 rows of that sub-setted data frame. We can write a snippet, as shown:

```
subdata_first_50=data[['Account Length','VMail Message','Day Calls']][1:50]
subdata_first_50
```

We can use the already created `subdata` data frame and subset it for the first 50 rows by typing:

```
subdata[1:50] or subdata[:50]
```

Alternatively, one can subset the columns using the list name as explained earlier and then subset for rows.

Another effective (but a little unstable, as its behaviour changes based on the version of Python installed) method to select both rows and columns together is the `.ix` method. Let's see how to use this method.

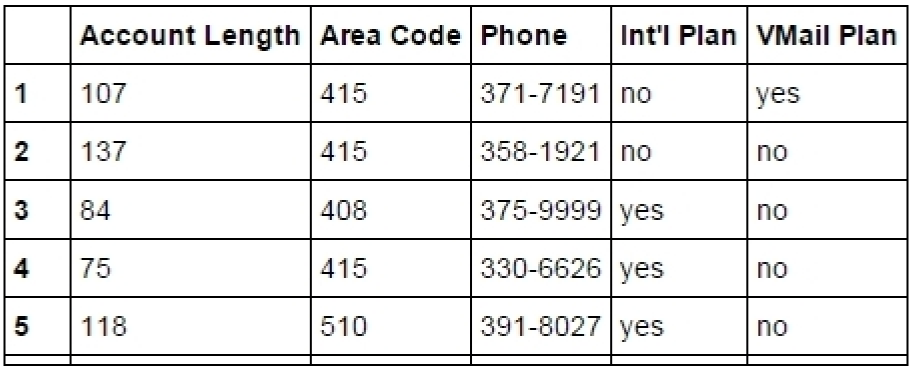
Basically, in the `.ix` method, we can provide row and column indices (in a lay man's term, row and column numbers) inside the square bracket. The syntax can be summarized, as follows:

- The data frame name is appended with `ix`
- Inside the square bracket, specify the row number (range) and column number (range) in that order

Now, let's have a look at a few examples:

- Selecting the first 100 rows of the first 5 columns:
- `data.ix[1:100,1:6]`

The output looks similar to the following screenshot:



	Account Length	Area Code	Phone	Int'l Plan	VMail Plan
1	107	415	371-7191	no	yes
2	137	415	358-1921	no	no
3	84	408	375-9999	yes	no
4	75	415	330-6626	yes	no
5	118	510	391-8027	yes	no

First 100 rows of the first 5 columns

- Selecting all rows from the first five columns:

```
data.ix[:,1:6]
```

- Selecting first 100 rows from all the columns:

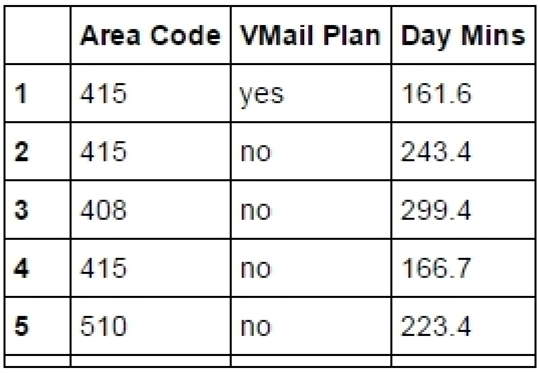
```
data.ix[1:100,:]
```

The row and column numbers/name can be passed off as a list, as well. Let's have a look at how it can be done:

- Selecting the first 100 rows from the 2nd, 5th, and 7th columns:

```
data.ix[1:100,[2,5,7]]
```

The output looks similar to the following screenshot:



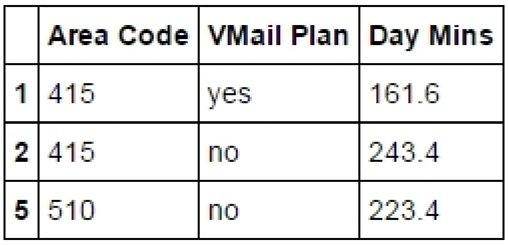
	Area Code	VMail Plan	Day Mins
1	415	yes	161.6
2	415	no	243.4
3	408	no	299.4
4	415	no	166.7
5	510	no	223.4

First 100 rows of the 2nd, 5th and 7th columns

- Selecting the 1st, 2nd and 5th rows from the 2nd, 5th and 7th columns:

```
data.ix[[1,2,5],[2,5,7]]
```

The output looks similar to the following screenshot:



	Area Code	VMail Plan	Day Mins
1	415	yes	161.6
2	415	no	243.4
5	510	no	223.4

1st, 2nd and 5th rows of the 2nd, 5th and 7th columns

Instead of row and column indices or numbers, we can also write corresponding column names, as shown in the following example:

```
data.ix[[1,2,5],['Area Code','VMail Plan','Day Mins']]
```

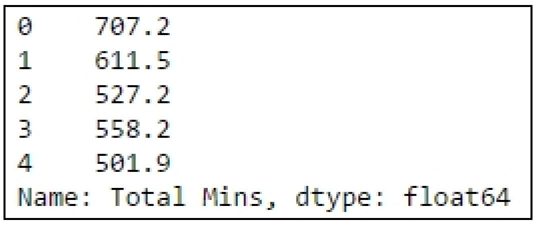
Creating new columns

Many times during the analysis, we are required to create a new column based on some calculation or modification of the existing columns containing a constant value to be used in the modelling. Hence, the knowledge of creating new columns becomes an indispensable tool to learn. Let's see how to do that.

Suppose, in the `Customer Churn Model` dataset, we want to calculate the total minutes spent during the day, evening, and night. This requires summing up the 3 columns, which are `Day Mins`, `Eve Mins`, and `Night Mins`. It can be done, as shown in the following snippet:

```
data['Total Mins']=data['Day Mins']+data['Eve Mins']+data['Night Mins']
data['Total Mins'].head()
```

The output of the snippet is, as follows:



```
0    707.2
1    611.5
2    527.2
3    558.2
4    501.9
Name: Total Mins, dtype: float64
```

First few entries of the new Total Mins column

Generating random numbers and their usage

Random numbers are just like any other number in their property except for the fact that they assume a different value every time the `call` statement to generate a random number is executed. Random number generating methods use certain algorithms to generate different numbers every time. However, after a finitely large period, they might start generating the already generated numbers. In that sense, these numbers are not truly random and are sometimes called pseudo-random numbers.

In spite of them actually being pseudo-random, these numbers can be assumed to be random for all practical purposes. These numbers are of critical importance to predictive analysts because of the following points:

- They allow analysts to perform simulations for probabilistic multicase scenarios
- They can be used to generate dummy data frames or columns of a data frame that are needed in the analysis
- They can be used for the random sampling of data

Various methods for generating random numbers

The method used to deal with random number is called `random` and is found in the `numpy` library. Let's have a look at the different methods of generating random numbers and their usage.

Let's start by generating a random integer between `1` and `100`. This can be done, as follows:

```
import numpy as np
np.random.randint(1,100)
```

If you run the preceding snippet, it will generate a random number between `1` and `100`. When I ran it, it gave me 43 as the result. It might give you something else.

To generate a random number between `0` and `1`, we can write something similar to the following code:

```
import numpy as np
np.random.random()
```

These methods allow us to generate one random number at a time. What if we wanted to generate a list of numbers, all lying within a given interval and generated randomly. Let's define a function that can generate a list of `n` random numbers lying between `a` and `b`.

All one needs to do is define a function, wherein an empty list is created and the randomly generated numbers are appended to the list. The recipe to do that is shown in the following code snippet:

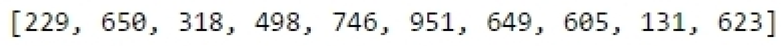
```
def randint_range(n,a,b):
    x=[]
    for i in range(n):
        x.append(np.random.randint(a,b))
    return x
```

NOTE: there are lots of tutorials on functions in python, for example https://www.learnpython.org/en/Functions

After defining this function we can generate, let's say, `10` numbers lying between `2` and `1000`, `as shown:`

```
rand_int_gen(10,2,1000)
```

On the first run, it gives something similar to the following output:



```
[229, 650, 318, 498, 746, 951, 649, 605, 131, 623]
```

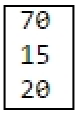
10 random integers between 2 and 1000

The `randrange` method is an important method to generate random numbers and is in a way an extension to the `randint` method, as it provides a step argument in addition to the start and stop argument in the case of `randint` function.

To generate three random numbers between `0` and `100,` which are all multiples of `5`, we can write:

```
import random
for i in range(3):
    print random.randrange(0,100,5)
```

You should get something similar to the following screenshot, as a result (the actual numbers might change):

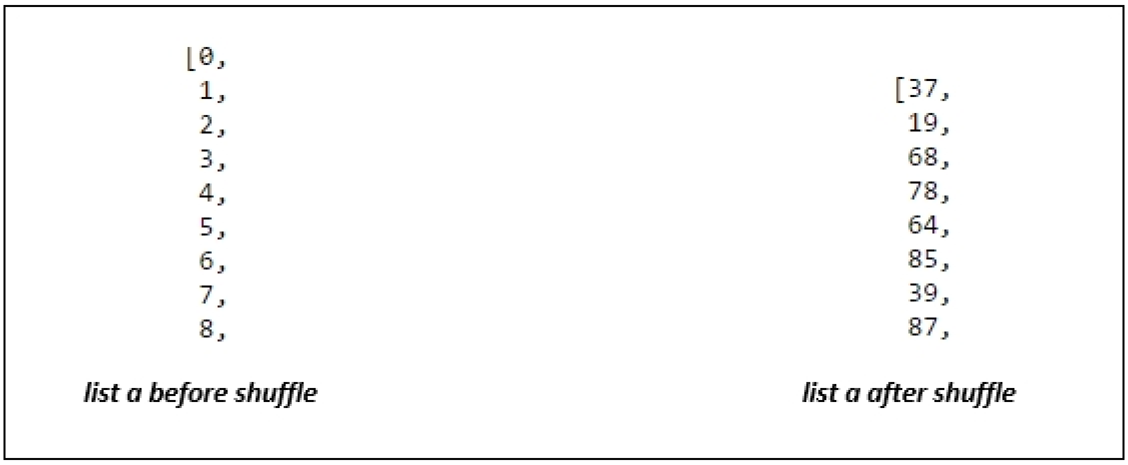


```
70
15
20
```

Another related useful method is `shuffle`, which shuffles a list or an array in random order. It doesn't generate a random number, per se, but nevertheless it is very useful. Lets see how it works. Lets generate a list of consecutive `100` integers and then shuffle the list:

```
a=range(100)
np.random.shuffle(a)
```

The list looks similar to the following screenshot before and after the shuffle:



```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
```

```
[37,
 19,
 68,
 78,
 64,
 85,
 39,
 87,
```

The `choice` method is another important technique that might come in very handy in various scenarios including creating simulations, depending upon selecting a random item from a list of items. The `choice` method is used to pick an item at random from a given list of items.

To see an example of how this method works, let's go back to the data frame that we have been using all along in this lab. Let's import that data again and get the list of column names, using the following code snippet:

```
import pandas as pd
data=pd.read_csv('C:/FILE_PATH/Customer Churn Model.txt')
column_list=data.columns.values.tolist()
```

To select one column name from the list, at random, we can write it similar to the following example:

```
np.random.choice(column_list)
```

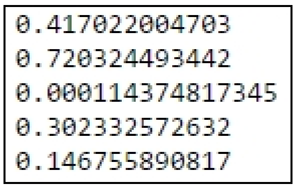
This should result in one column name being chosen at random from the list of the column names. I got `Day Calls` for my run. Of course, one can loop over the choice method to get multiple items, as we did for the `randint` method.

Seeding a random number

At the onset of this section on random numbers, we discussed how random numbers change their values on every execution of their call statement. They repeat their values but only after a very large period. Sometimes, we need to generate a set of random numbers that retain their value. This can be achieved by seeding the generation of random numbers. Basically, the particular instance of generating a random number is given a seed (sort of a key), which when used can regenerate the same set of random numbers. Let's see this with an example:

```
np.random.seed(1)
for i in range(5):
    print np.random.random()
```

In the first line, we set the seed as `1` and then generated `5` random numbers. The output looks something similar to this:



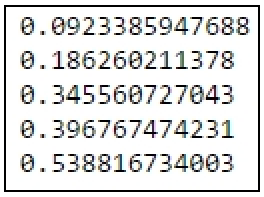
```
0.417022004703
0.720324493442
0.000114374817345
0.302332572632
0.146755890817
```

Five random numbers generated through random method with seed 1

If one removes the seed and then generates random numbers, one will get different random numbers. Let's have a look:

```
for i in range(5):
    print np.random.random()
```

By running the preceding code snippet, one indeed gets different random numbers, as shown in the following output screenshot:



```
0.0923385947688
0.186260211378
0.345560727043
0.396767474231
0.538816734003
```

Five random number generated through random method without seed 1

However, if one brings back the seed used to generate random numbers, we can get back the same numbers. If we try running the following snippet, we will have to regenerate the numbers, as shown in the first case:

```
np.random.seed(1)
for i in range(5):
    print np.random.random()
```

Generating random numbers following probability distributions

If you have studied probability in the past you might have heard of probability distributions. There are two concepts that you might want to refresh.

Probability density function

For a random variable, it is just the count of times that the random variable attains a particular value `x` or the number of times that the value of the random variable falls in a given range (bins). This gives the probability of attaining a particular value by the random variable. Histograms plot this number/probability on the *y* axis and it can be identified as the *y* axis value of a distribution plot/histogram:

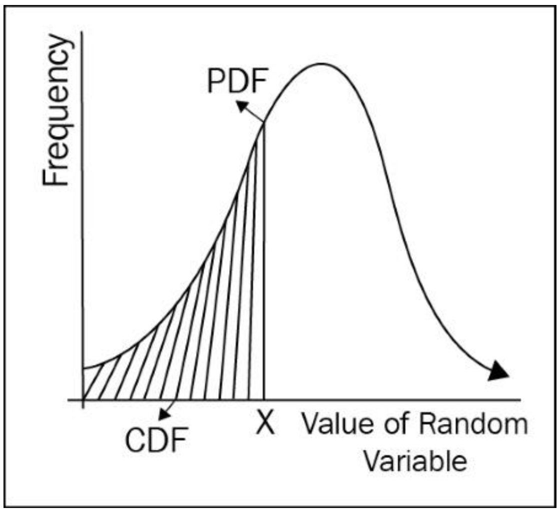
```
PDF = Prob(X=x)
```

Cumulative density function

For a random variable, it is defined as the probability that the random variable is less than or equal to a given value `x`. It is the total probability that the random variable is less than or equal to a given value. For a given point on the *x* axis, it is calculated as the area enclosed by the frequency distribution curve between by values less than `x`.

Mathematically, it is defined as follows:

```
CDF(x) = Prob(X<=x)
```



CDF is the area enclosed by the curve till that value of random variable. PDF is the frequency/probability of that particular value of random variable.

There are various kinds of probability distributions that frequently occur, including the normal (famously known as the **Bell Curve**), uniform, poisson, binomial, multinomial distributions, and so on.

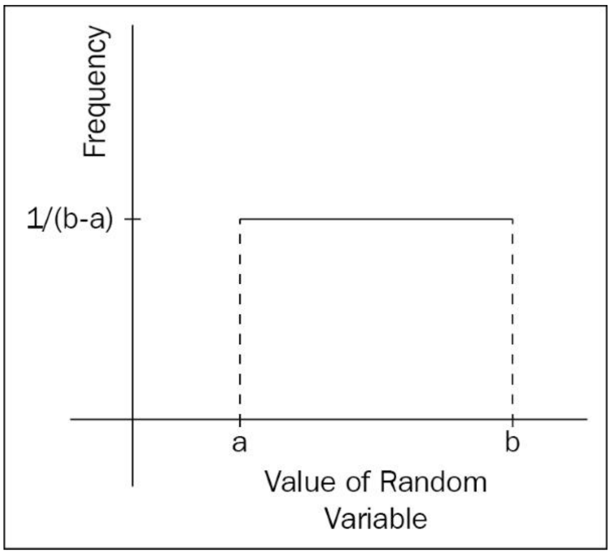
Many of the analyses require generating random numbers that follow a particular probability distribution. One can generate random numbers in such a fashion using the same `random` method of the `numpy` library.

Let's see how one can generate two of the most commonly used distributions, which are normal and uniform distributions.

Uniform distribution

A uniform distribution is defined by its endpoints—the start and stop points. Each of the points lying in between these endpoints are supposed to occur with the same (uniform) probability and hence the name of the distribution.

If the start and stop points are **a** and **b**, each point between **a** and **b** would occur with a frequency of **1/(b-a)**:



In a uniform distribution, all the random variables occur with the same (uniform) frequency/probability

As the uniform distribution is defined by its start and stop points, it is essential to know these points while generating random numbers following a uniform distribution. Thus, these points are taken as input parameters for the uniform function that is used to generate a random number following a uniform distribution. The other parameter of this function is the number of random numbers that one wants to generate.

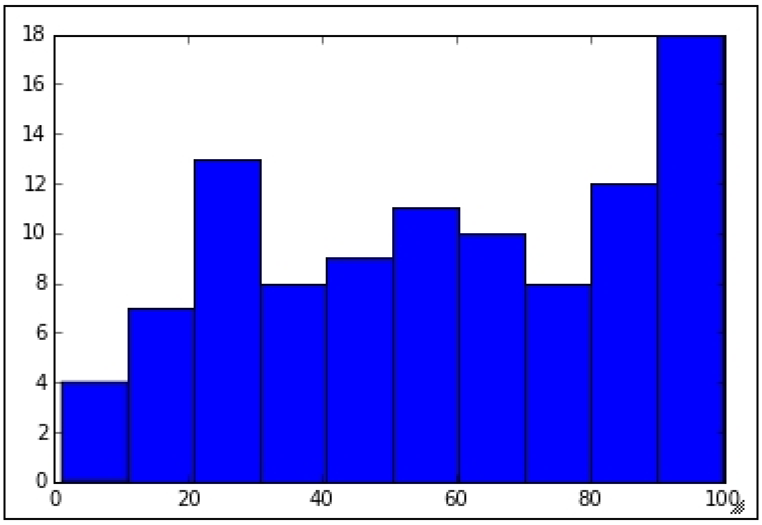
To generate 100 random numbers lying between `1` and `100`, one can write the following:

```
import numpy as np
randnum=np.random.uniform(1,100,100)
```

To check whether it indeed follows the uniform distribution, let's plot a histogram of these numbers and see whether they occur with the same probability or not. This can be done using the following code snippet:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
a=np.random.uniform(1,100,100)
b=range(1,101)
plt.hist(a)
```

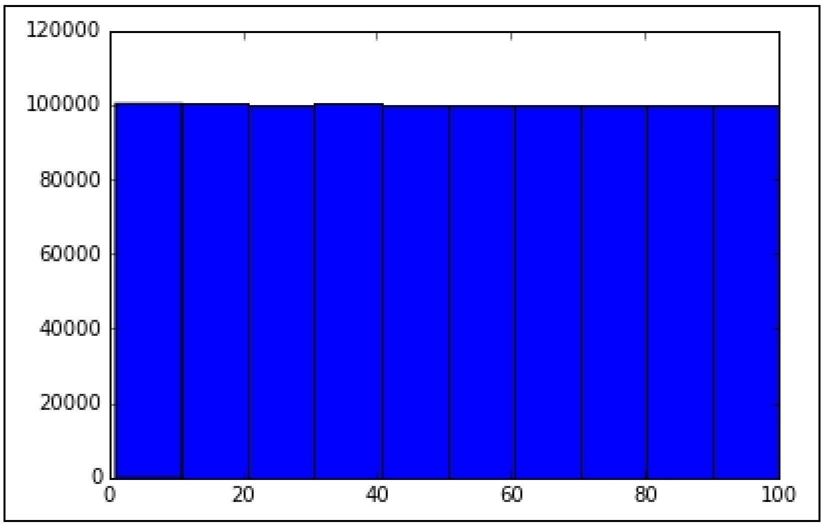
The output that we get is not what we expected. It doesn't have the same probability for all the numbers, as seen in the following output:



Histogram of 100 random numbers between 1 and 100 following uniform distribution

The reason for this is that 100 is a very small number, given the range (1-100), to showcase the property of the uniform distribution. We should try generating more random numbers and then see the results. Try generating around a million (1,000,000) numbers by changing the parameter in the `uniform` function, and then see the results of the preceding code snippet.

It should look something like the following:



The kind of plot expected for uniform distribution, all the numbers occur with the same frequency/probability

If you observe the preceding plot properly, each bin that contains 10 numbers occurs roughly with a frequency of 100,000 (and hence a probability of *100000/1000000=1/10*). This means that each number occurs with a probability of *1/10*1/10=1/100*, which is equal to the probability that we would have expected from a set of numbers following the uniform distribution between 1 and 100 *(1/(100-1)=1/99)*.

Normal distribution

Normal distribution is the most common form of probability distribution arising from everyday real-life situations. Thus, the exam score distribution of students in a class would roughly follow the normal distribution as would the heights of the students in the class. An interesting behavior of all the probability distributions is that they tend to follow/align to a normal distribution as the sample size of the numbers increase. In a sense, one can say that a normal distribution is the most ubiquitous and versatile probability distribution around.

The parameters that define a normal distribution are the mean and standard deviation. A normal distribution with a `0` mean and `1` standard deviation is called a standard normal distribution. The `randn` function of the `random` method is used to generate random numbers following a normal distribution. It returns random numbers following a standard normal distribution.

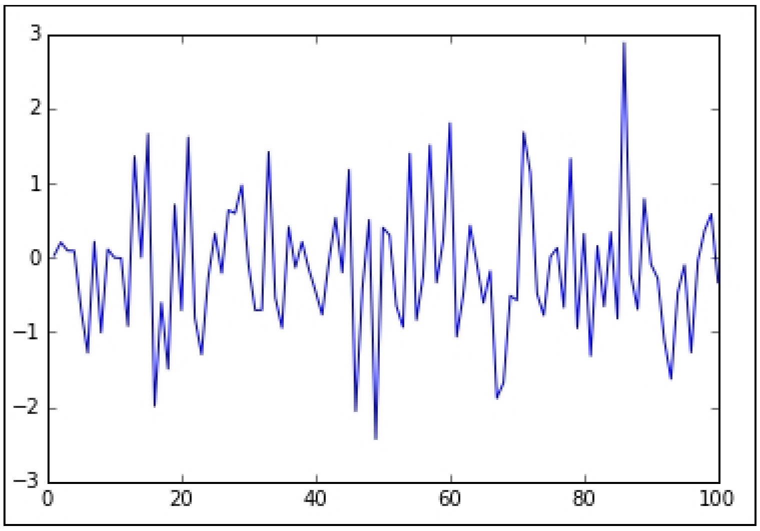
To generate `100` such numbers, one simply writes the following:

```
import numpy as np
a=np.random.randn(100)
```

To take a look at how random these values actually are, let's plot them against a list of integers:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
a=np.random.randn(100)
b=range(1,101)
plt.plot(b,a)
```

The output looks something like the following image. The numbers are visibly random.



A plot of 100 random numbers following normal distribution

One can pass a list defining the shape of the expected array. If one passes, let's say, `(2,4)` as the input, one would get a 2 x 4 array of numbers following a standard normal distribution:

```
import numpy as np
a=np.random.randn(2,4)
```

If no numbers are specified, it generates a single random number from the standard normal distribution.

To get numbers following normal distributions (with mean and standard deviation other than `0` and `1`, let's say, mean `1.5` and standard deviation `2.5`), one can write something like the following:

```
import numpy as np
a=2.5*np.random.randn(100)+1.5
```

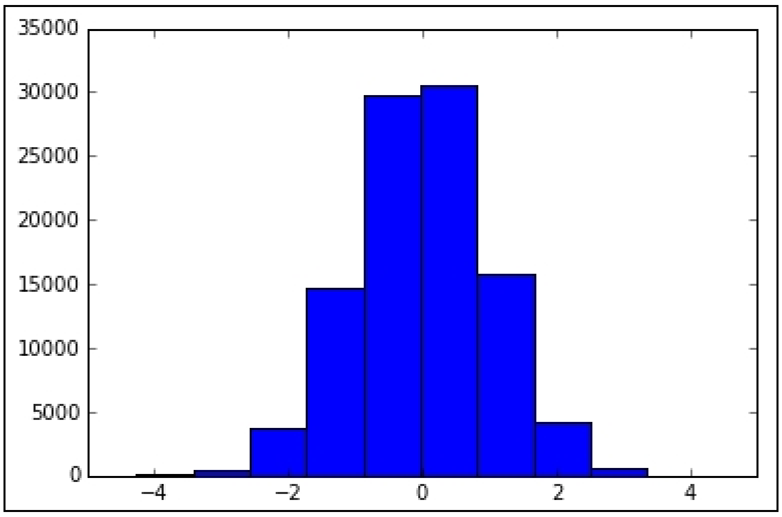
The preceding calculation holds because the standard normal distribution *S* is created from a normal distribution *X*, with mean μ and standard deviation σ, using the following formula:

$$S = (X - \mu) / \sigma$$

Let's generate enough random numbers following a standard normal distribution and plot them to see whether they follow the shape of a standard normal distribution (a bell curve). This can be done using the following code snippet:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
a=np.random.randn(100000)
b=range(1,101)
plt.hist(a)
```

The output would look something like this, which roughly looks like a bell curve (if one joins the top points of all the bins to form a curvilinear line):



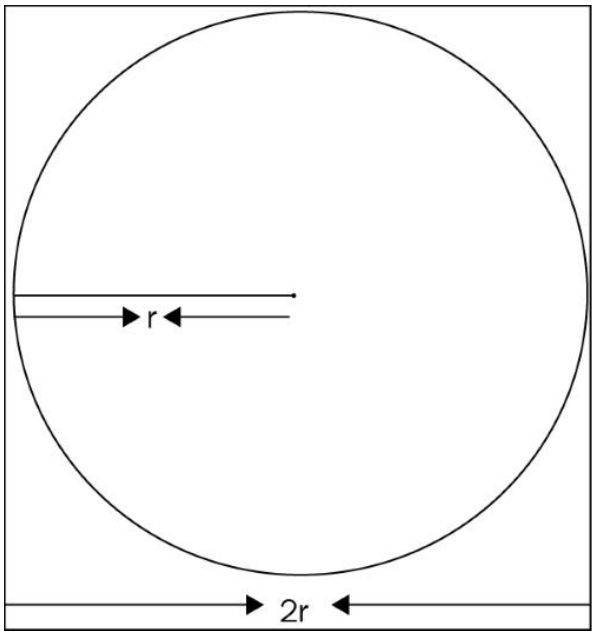
Histogram of 100000 random numbers following standard normal distribution.

Using the Monte-Carlo simulation to find the value of pi

So far, we have been learning about various ways to generate random numbers. Let's now see an application of random numbers. In this section, we will use random numbers to run something called Monte-Carlo simulations to calculate the value of pi. These simulations are based on repeated random sampling or the generation of numbers.

Geometry and mathematics behind the calculation of pi

Consider a circle of radius r unit circumscribed inside a square of side **2r** units such that the circle's diameter and the square's sides have the same dimensions:



A circle of radius r circumscribed in a square of side 2r

What is the probability that a point chosen at random would lie inside the circle? This probability would be given by the following formulae:

$$Prob(point\ lying\ inside\ the\ circle) = Area\ of\ circle / Area\ of\ square$$

$$p = pi * r * r / (2r * 2r)$$

$$p = pi / 4$$

Thus, we find out that the probability of a point lying inside the circle is pi/4. The purpose of the simulation is to calculate this probability and use this to estimate the value of pi. The following are the steps to be implemented to run this simulation:

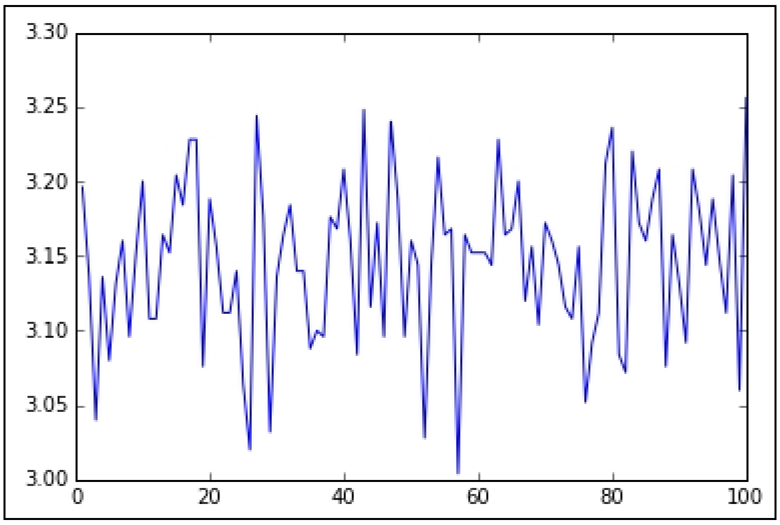
1. Generate points with both *x* and *y* coordinates lying between `0` and `1`.
2. Calculate *x*x + y*y*. If it is less than `1`, it lies inside the circle. If it is greater than 1, it lies outside the circle.
3. Calculate the total number of points that lie inside the circle. Divide it by the total number of points generated to get the probability of a point lying inside the circle.
4. Use this probability to calculate the value of pi.

5. Repeat the process for a sufficient number of times, say, 1,000 times and generate 1,000 different values of pi.
6. Take an average of all the 1,000 values of pi to arrive at the final value of pi.

Let's see how one can implement these steps in Python. The following code snippet would do just this:

```
pi_avg=0
pi_value_list=[]
for i in range(100):
    value=0
    x=np.random.uniform(0,1,1000).tolist()
    y=np.random.uniform(0,1,1000).tolist()
    for j in range(1000):
            z=np.sqrt(x[j]*x[j]+y[j]*y[j])
            if z<=1:
                value+=1
    float_value=float(value)
    pi_value=float_value*4/1000
    pi_value_list.append(pi_value)
    pi_avg+=pi_value

pi=pi_avg/100
print pi
ind=range(1,101)
fig=plt.plot(ind,pi_value_list)
fig
```

The preceding snippet generates 1,000 random points to calculate the probability of a point lying inside the circle and then repeats this process 100 times to get at the final averaged value of pi. These 100 values of pi have been plotted and they look as follows:



Values of pi over 100 simulations of 1000 points each

The final averaged value of pi comes out to be `3.14584` in this run. As we increase the number of runs, the accuracy increases. One can easily wrap the preceding snippet in a function and pass the number of runs as an input for the easy comparison of pi values as an increasing number of runs are passed to this function. The following code snippet is a function to do just this:

```
def pi_run(nums,loops):
    pi_avg=0
    pi_value_list=[]
    for i in range(loops):
        value=0
        x=np.random.uniform(0,1,nums).tolist()
        y=np.random.uniform(0,1,nums).tolist()
        for j in range(nums):
            z=np.sqrt(x[j]*x[j]+y[j]*y[j])
            if z<=1:
                value+=1
        float_value=float(value)
        pi_value=float_value*4/nums
        pi_value_list.append(pi_value)
        pi_avg+=pi_value

    pi=pi_avg/loops
    ind=range(1,loops+1)
    fig=plt.plot(ind,pi_value_list)
    return (pi,fig)
```

To call this function, write `pi_run(1000,100)`, and it should give you a similar result as was given previously with the hardcoded numbers. This function would return both the averaged value of pi as well as the plot.

Generating a dummy data frame

One very important use of generating random numbers is to create a dummy data frame, which will be used extensively to illustrate concepts and examples.

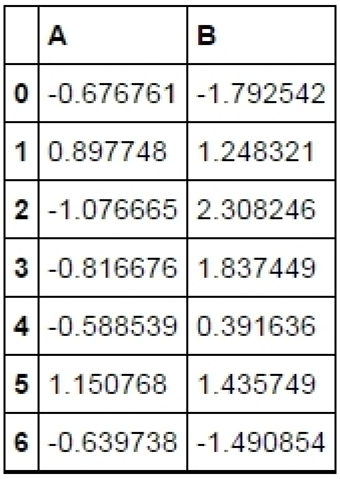
The basic concept of this is that the array/list of random numbers generated through the various methods described in the previous sections can be passed as the columns of a data frame. The column names and their descriptions are passed as the keys and values of a dictionary.

Let's see an example where a dummy data frame contains two columns, `A` and `B`, which have `10` random numbers following a standard normal distribution and normal distribution, respectively.

To create such a data frame, one can run the following code snippet:

```
import pandas as pd
d=pd.DataFrame({'A':np.random.randn(10),'B':2.5*np.random.randn(10)+1.5})
d
```

The following screenshot is the output of the code:



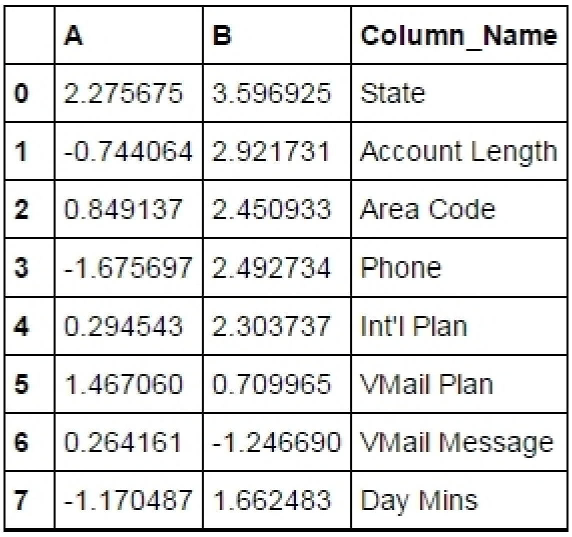
	A	B
0	-0.676761	-1.792542
1	0.897748	1.248321
2	-1.076665	2.308246
3	-0.816676	1.837449
4	-0.588539	0.391636
5	1.150768	1.435749
6	-0.639738	-1.490854

A dummy data frame containing 2 columns – one having numbers following standard normal distribution, the second having random numbers following normal distribution with mean 1.5 and standard deviation 2.5

Categorical/string variables can also be passed as a list to be part of a dummy data frame. Let's go back to our example of the `Customer Churn Model` data and use the column names as the list to be passed. This can be done as described in the following snippet:

```
import pandas as pd
data = pd.read_csv('C:/FILE_PATH/Customer Churn Model.txt')
column_list=data.columns.values.tolist()
a=len(column_list)
d=pd.DataFrame({'Column_Name':column_list,'A':np.random.randn(a),'B':2.5*np
.random.randn(a)+1.5})
d
```

The output of the preceding snippet is as follows:



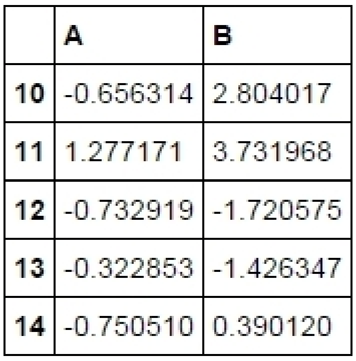
	A	B	Column_Name
0	2.275675	3.596925	State
1	-0.744064	2.921731	Account Length
2	0.849137	2.450933	Area Code
3	-1.675697	2.492734	Phone
4	0.294543	2.303737	Int'l Plan
5	1.467060	0.709965	VMail Plan
6	0.264161	-1.246690	VMail Message
7	-1.170487	1.662483	Day Mins

Another dummy data frame. Similar to the one above but with one extra column which has column names of the data data frame

The index can also be passed as one of the parameters of this function. By default, it gives a range of numbers starting from `0` as the index. If we want something else as the index, we can specify it in the index parameter as shown in the following example:

```
import pandas as pd
d=pd.DataFrame({'A':np.random.randn(10),'B':2.5*np.random.randn(10)+1.5},in
dex=range(10,20))
d
```

The output of the preceding code looks like the following:



	A	B
10	-0.656314	2.804017
11	1.277171	3.731968
12	-0.732919	-1.720575
13	-0.322853	-1.426347
14	-0.750510	0.390120

Passing indices to the dummy data frame

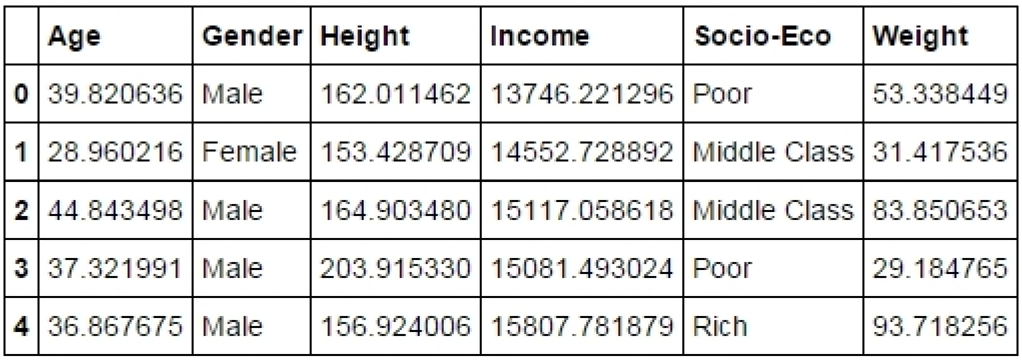
Grouping the data – aggregation, filtering, and transformation

In this section, you will learn how to aggregate data over categorical variables. This is a very common practice when the data consists of categorical variables. This analysis enables us to conduct a category-wise analysis and take further decisions regarding the modelling.

To illustrate the concepts of grouping and aggregating data better, let's create a simple dummy data frame that has a rich mix of both numerical and categorical variables. Let's use whatever we have explored upto now about random numbers to create this data frame, as shown in the following snippet:

```
import numpy as np
import pandas as pd
a=['Male','Female']
b=['Rich','Poor','Middle Class']
gender=[]
seb=[]
for i in range(1,101):
    gender.append(np.random.choice(a))
    seb.append(np.random.choice(b))
height=30*np.random.randn(100)+155
weight=20*np.random.randn(100)+60
age=10*np.random.randn(100)+35
income=1500*np.random.randn(100)+15000

df=pd.DataFrame({'Gender':gender,'Height':height,'Weight':weight,'Age':age,
'Income':income,'Socio-Eco':seb})
df.head()
```

The output data frame `df` looks something as follows:



	Age	Gender	Height	Income	Socio-Eco	Weight
0	39.820636	Male	162.011462	13746.221296	Poor	53.338449
1	28.960216	Female	153.428709	14552.728892	Middle Class	31.417536
2	44.843498	Male	164.903480	15117.058618	Middle Class	83.850653
3	37.321991	Male	203.915330	15081.493024	Poor	29.184765
4	36.867675	Male	156.924006	15807.781879	Rich	93.718256

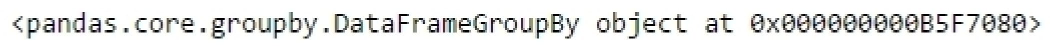
The resulting dummy data frame df containing 6 columns

As we can see from the preceding code snippet, the shape of the data frame is 100x6.

Grouping can be done over a categorical variable using the `groupby` function. The column name of the categorical variable needs to be specified for this. Suppose that we wish to group the data frame based on the `Gender` variable. This can be done by writing the following:

```
df.groupby('Gender')
```

If you run the preceding snippet on your IDE, you will get the following output indicating that a `groupby` object has been created:



```
<pandas.core.groupby.DataFrameGroupBy object at 0x000000000B5F7080>
```

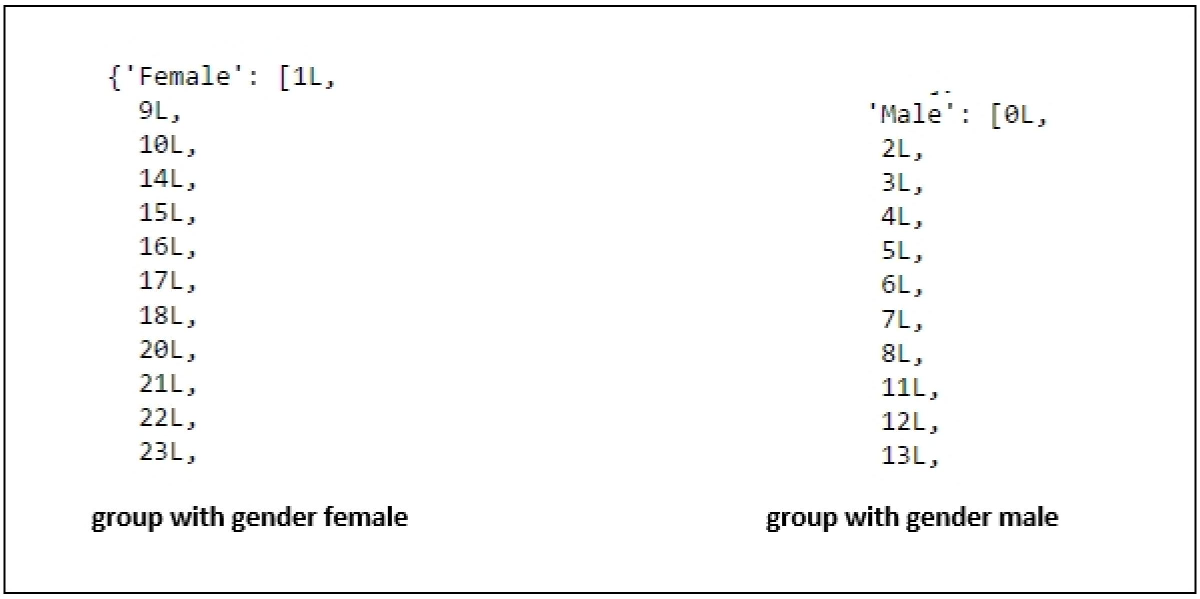
Prompt showing that the groupby object has been created

The `groupby` function doesn't split the original data frame into several groups, instead it creates a `groupby` object that has two attributes, which are `name` and `group`.

These attributes can be accessed by following the name of the `groupby` object with `'.'`, followed by the name of the attribute. For example, to access the group attribute, one can write the following:

```
grouped = df.groupby('Gender')
grouped.groups
```

The following is the output:


```
{'Female': [1L,
  9L,
  10L,
  14L,
  15L,
  16L,
  17L,
  18L,
  20L,
  21L,
  22L,
  23L,
```

group with gender female

```
'Male': [0L,
  2L,
  3L,
  4L,
  5L,
  6L,
  7L,
  8L,
  11L,
  12L,
  13L,
```

group with gender male

Two groups based on gender

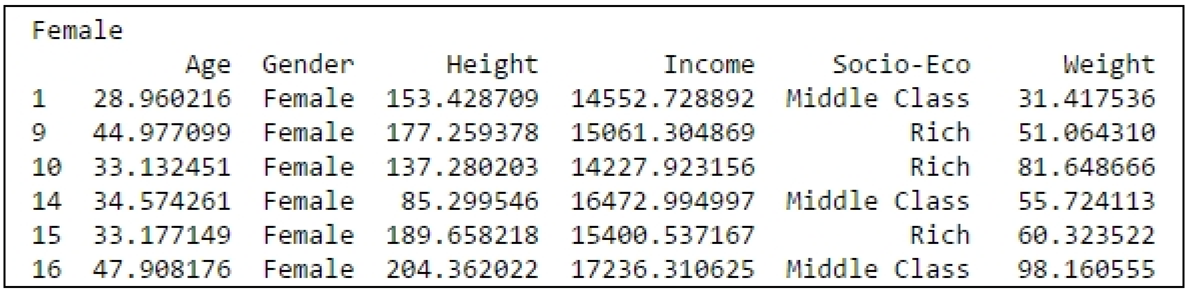
The numbers indicate the row numbers that belong to that particular group.

One important feature of these attributes is that they are iterable, and the same operation can be applied to each group just by looping. This comes in very handy when the number of groups are large and one needs results of the operation separately for each group.

Let's perform a simple operation to illustrate this. Let's try to print the name and groups in the `groupby` object that we just created. This can be done as follows:

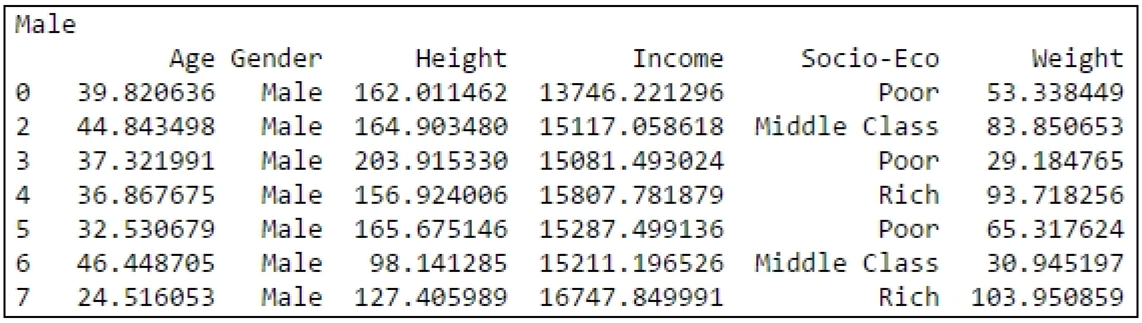
```
grouped=df.groupby('Gender')
for names,groups in grouped:
    print names
    print groups
```

This prints the name of the group followed by the entire data for this group. The output looks something like the following:


```
Female
          Age  Gender      Height        Income     Socio-Eco     Weight
1   28.960216  Female  153.428709  14552.728892  Middle Class  31.417536
9   44.977099  Female  177.259378  15061.304869          Rich  51.064310
10  33.132451  Female  137.280203  14227.923156          Rich  81.648666
14  34.574261  Female   85.299546  16472.994997  Middle Class  55.724113
15  33.177149  Female  189.658218  15400.537167          Rich  60.323522
16  47.908176  Female  204.362022  17236.310625  Middle Class  98.160555
```

Name and the data in the group with gender female

Here is the second group as a part of the output:



```
Male
          Age Gender      Height        Income     Socio-Eco      Weight
0   39.820636   Male  162.011462  13746.221296          Poor   53.338449
2   44.843498   Male  164.903480  15117.058618  Middle Class   83.850653
3   37.321991   Male  203.915330  15081.493024          Poor   29.184765
4   36.867675   Male  156.924006  15807.781879          Rich   93.718256
5   32.530679   Male  165.675146  15287.499136          Poor   65.317624
6   46.448705   Male   98.141285  15211.196526  Middle Class   30.945197
7   24.516053   Male  127.405989  16747.849991          Rich  103.950859
```

Name and the data in the group with gender male

A single group can be selected by writing the following:

```
grouped.get_group('Female')
```

This would generate only the first of the two groups, as shown in the preceding screenshot.

A data frame can be grouped over more than one categorical variable as well. As in this case, the data frame can be grouped over both `Gender` and `Soci-Eco` by writing something like the following:

```
grouped=df.groupby(['Gender','Socio-Eco'])
```

This should create six groups from a combination of two categories of `Gender` and three categories of the `Socio-Eco` variable. This can be checked by checking the length of the `groupby` object as follows:

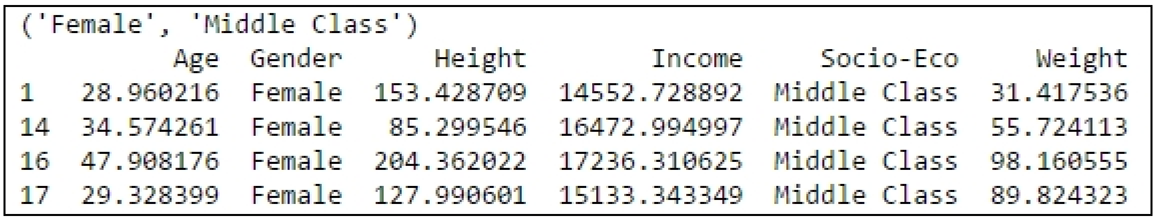
```
len(grouped)
```

It indeed returns six. To look at how these groups look, let's run the same iteration on the group attributes as we did earlier:

```
grouped=df.groupby(['Gender','Socio-Eco'])
for names,groups in grouped:
    print names
    print groups
```

The code gives six groups' names and their entire data as the output. There would be six of such groups in total.

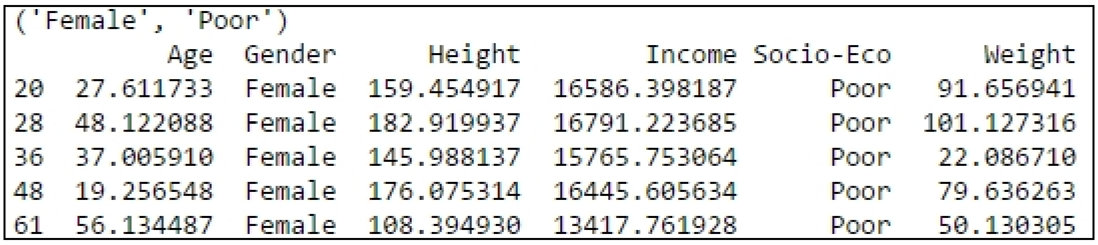
The first group looks like the following:



```
('Female', 'Middle Class')
          Age  Gender      Height        Income     Socio-Eco     Weight
1   28.960216  Female  153.428709  14552.728892  Middle Class  31.417536
14  34.574261  Female   85.299546  16472.994997  Middle Class  55.724113
16  47.908176  Female  204.362022  17236.310625  Middle Class  98.160555
17  29.328399  Female  127.990601  15133.343349  Middle Class  89.824323
```

Name and the data in the group with gender female and Socio_Eco Middle Class

The second group looks like the following:



```
('Female', 'Poor')
          Age  Gender      Height        Income Socio-Eco      Weight
20  27.611733  Female  159.454917  16586.398187      Poor   91.656941
28  48.122088  Female  182.919937  16791.223685      Poor  101.127316
36  37.005910  Female  145.988137  15765.753064      Poor   22.086710
48  19.256548  Female  176.075314  16445.605634      Poor   79.636263
61  56.134487  Female  108.394930  13417.761928      Poor   50.130305
```

Name and the data in the group with gender female and Socio_Eco Middle Class

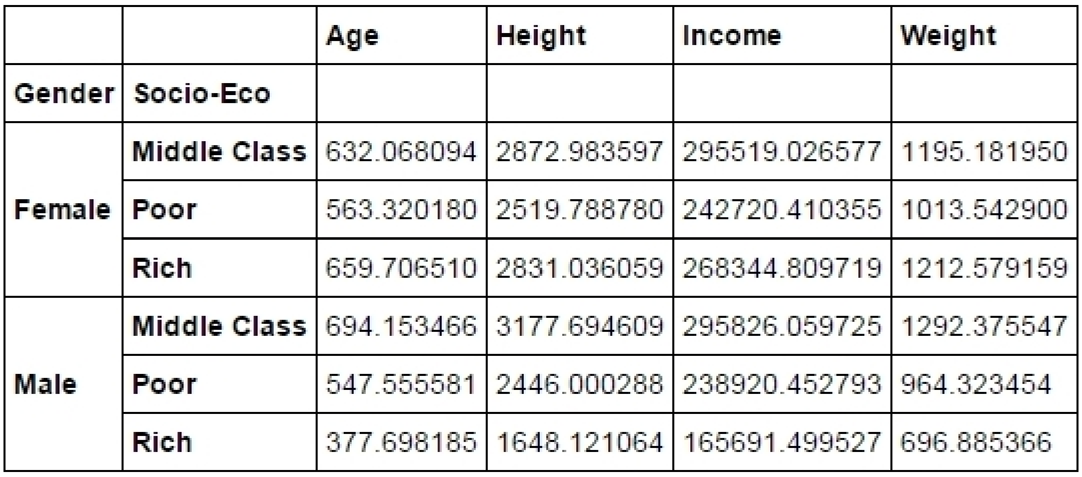
Aggregation

There are various aggregations that are possible on a data frame, such as `sum`, `mean`, `describe`, `size`, and so on. The aggregation basically means applying a function to all the groups all at once and getting a result from that particular group.

Let's see the `sum` function. We just need to write the following code snippet to see how it works:

```
grouped=df.groupby(['Gender','Socio-Eco'])
grouped.sum()
```

We gets the following table as the result:



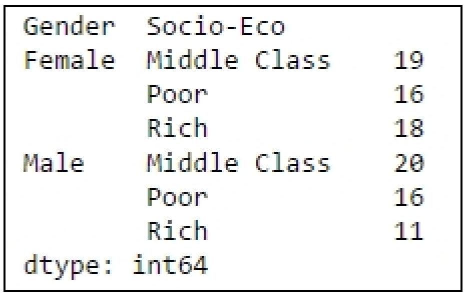
		Age	Height	Income	Weight
Gender	**Socio-Eco**				
Female	**Middle Class**	632.068094	2872.983597	295519.026577	1195.181950
	Poor	563.320180	2519.788780	242720.410355	1013.542900
	Rich	659.706510	2831.036059	268344.809719	1212.579159
Male	**Middle Class**	694.153466	3177.694609	295826.059725	1292.375547
	Poor	547.555581	2446.000288	238920.452793	964.323454
	Rich	377.698185	1648.121064	165691.499527	696.885366

Sum of each column for different groups

To get the number of rows in each group (or calculate the size of each group), we can write something similar to the following code snippet:

```
grouped=df.groupby(['Gender','Socio-Eco'])
grouped.size()
```

This results in a table, as shown in the following screenshot:



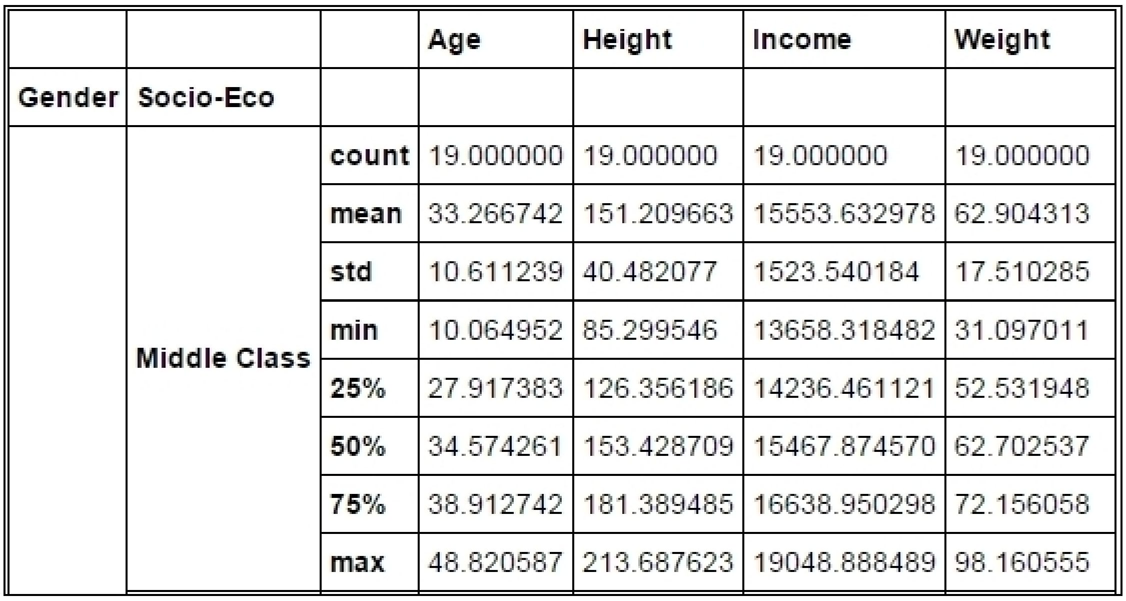
```
Gender  Socio-Eco
Female  Middle Class    19
        Poor            16
        Rich            18
Male    Middle Class    20
        Poor            16
        Rich            11
dtype: int64
```

Size of each group

One can use the `describe` function to get the summary statistics for each group separately. The syntax is exactly the same as it is for the earlier two functions:

```
grouped=df.groupby(['Gender','Socio-Eco'])
grouped.describe()
```

This output looks similar to the following table:



			Age	Height	Income	Weight
Gender	**Socio-Eco**					
	Middle Class	**count**	19.000000	19.000000	19.000000	19.000000
		mean	33.266742	151.209663	15553.632978	62.904313
		std	10.611239	40.482077	1523.540184	17.510285
		min	10.064952	85.299546	13658.318482	31.097011
		25%	27.917383	126.356186	14236.461121	52.531948
		50%	34.574261	153.428709	15467.874570	62.702537
		75%	38.912742	181.389485	16638.950298	72.156058
		max	48.820587	213.687623	19048.888489	98.160555

All the summary statistics of each column for different groups

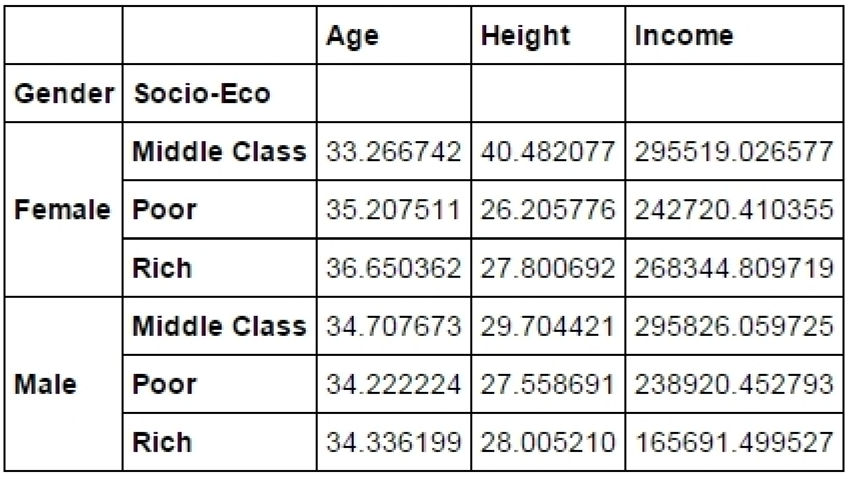
The `groupby` objects behave similar to an individual data frame, in the sense that one can select columns from these `groupby` objects just as we do from the data frames:

```
grouped=df.groupby(['Gender','Socio-Eco'])
grouped_income=grouped['Income']
```

One can apply different functions to different columns. The `aggregate` method used to do this is shown in the following snippet. With the following snippet, one can calculate `sum` of `Income`, `mean` of `Age`, and standard deviation of `Height`, as shown:

```
grouped=df.groupby(['Gender','Socio-Eco'])
grouped.aggregate({'Income':np.sum,'Age':np.mean,'Height':np.std})
```

The output of the preceding snippet looks similar to the following table:



		Age	Height	Income
Gender	Socio-Eco			
Female	Middle Class	33.266742	40.482077	295519.026577
	Poor	35.207511	26.205776	242720.410355
	Rich	36.650362	27.800692	268344.809719
Male	Middle Class	34.707673	29.704421	295826.059725
	Poor	34.222224	27.558691	238920.452793
	Rich	34.336199	28.005210	165691.499527

Selected summary statistics of selected columns for different groups

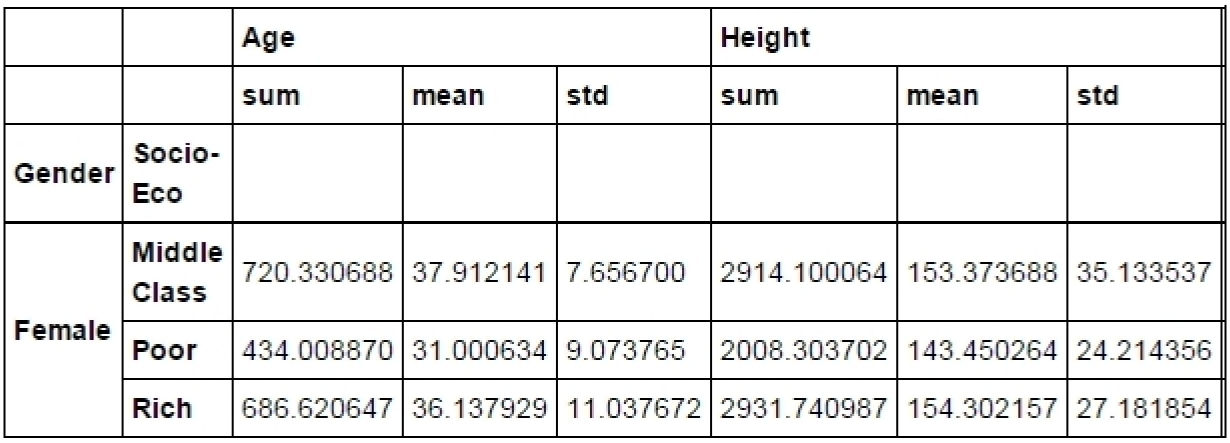
We can also define a function using the `lambda` method of defining a calculation in Python. Suppose you don't want the mean of age but the ratio of mean and standard deviation for height. You can define the formula for this ratio using the `lambda` method, illustrated as follows:

```
grouped=df.groupby(['Gender','Socio-Eco'])
grouped.aggregate({'Age':np.mean,'Height':lambda x:np.mean(x)/np.std(x)})
```

Rather than applying different functions to different columns, one can apply several functions to all the columns at the same time, as shown:

```
grouped.aggregate([np.sum, np.mean, np.std])
```

The output of the code snippet contains the result of all the three functions applied on all the columns of the `groupby` object, as seen in the following screenshot:



		Age			Height		
		sum	mean	std	sum	mean	std
Gender	Socio-Eco						
Female	Middle Class	720.330688	37.912141	7.656700	2914.100064	153.373688	35.133537
	Poor	434.008870	31.000634	9.073765	2008.303702	143.450264	24.214356
	Rich	686.620647	36.137929	11.037672	2931.740987	154.302157	27.181854

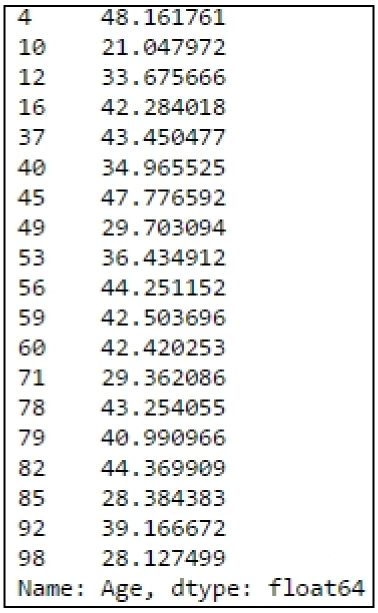
More than one selected summary statistics of selected columns for different groups

Filtering

One important operation that can be applied on the `groupby` objects is filter. We can filter elements based on the properties of groups. Suppose we want to choose elements from the `Age` column that are a part of the group wherein the sum of `Age` is greater than `700`. This filtering can be done by writing the following snippet:

```
grouped['Age'].filter(lambda x:x.sum()>700)
```

The output contains the row numbers that are part of the group where the sum of `Age` is greater than `700`. The output is, as follows:



```
4     48.161761
10    21.047972
12    33.675666
16    42.284018
37    43.450477
40    34.965525
45    47.776592
49    29.703094
53    36.434912
56    44.251152
59    42.503696
60    42.420253
71    29.362086
78    43.254055
79    40.990966
82    44.369909
85    28.384383
92    39.166672
98    28.127499
Name: Age, dtype: float64
```

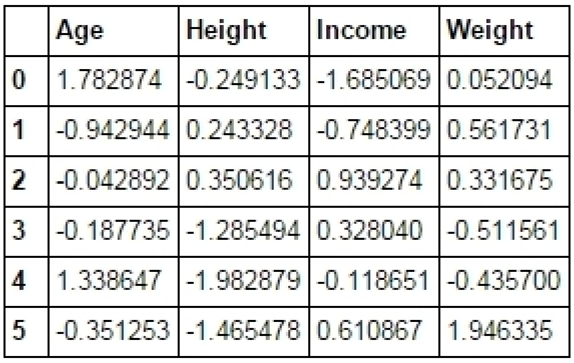
The rows left after filtering it for elements, which are part of groups, where the sum of ages is greater than 700

Transformation

One can use the `transform` method to mathematically transform all the elements in a numerical column. Suppose, we wish to calculate the standard normal values for all the elements in the numerical columns of our data frame; this can be done in a manner as shown:

```
zscore = lambda x: (x - x.mean()) / x.std()
grouped.transform(zscore)
```

The output contains standard normal values for all the numerical columns in the data frame, as shown in the following screenshot:



	Age	Height	Income	Weight
0	1.782874	-0.249133	-1.685069	0.052094
1	-0.942944	0.243328	-0.748399	0.561731
2	-0.042892	0.350616	0.939274	0.331675
3	-0.187735	-1.285494	0.328040	-0.511561
4	1.338647	-1.982879	-0.118651	-0.435700
5	-0.351253	-1.465478	0.610867	1.946335

Result of applying a lambda defined function on the columns of groups

The `transform` method comes in handy in a lot of situations. For example, it can be used to fill the missing values with the mean of the non-missing values, as shown:

```
f = lambda x: x.fillna(x.mean()
grouped.transform(f)
```

Miscellaneous operations

In many situations, one needs to select the *n*th row of each group of a `groupby` object, most often the first and the last row. This can be easily done once the `groupby` object is created. Let's see how:

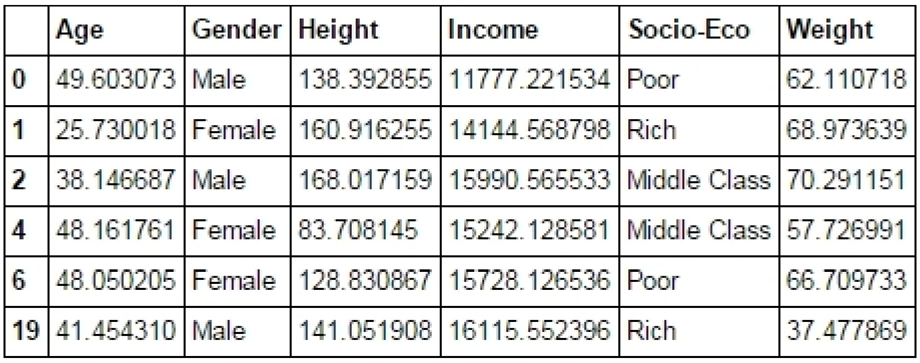
- The first row of each group can be selected by writing the following code snippet:

  ```
  grouped.head(1)
  ```

- While the last row of each group can be selected by writing the following code snippet:

  ```
  grouped.tail(1)
  ```

The result of the former, is as shown:



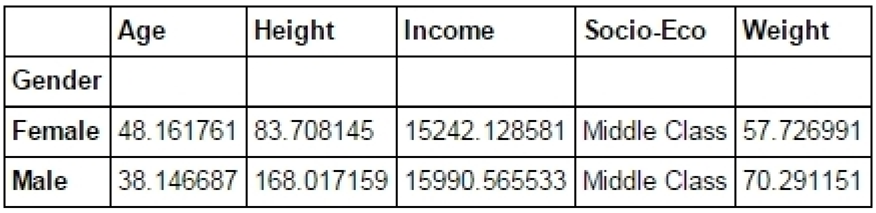
	Age	Gender	Height	Income	Socio-Eco	Weight
0	49.603073	Male	138.392855	11777.221534	Poor	62.110718
1	25.730018	Female	160.916255	14144.568798	Rich	68.973639
2	38.146687	Male	168.017159	15990.565533	Middle Class	70.291151
4	48.161761	Female	83.708145	15242.128581	Middle Class	57.726991
6	48.050205	Female	128.830867	15728.126536	Poor	66.709733
19	41.454310	Male	141.051908	16115.552396	Rich	37.477869

First few rows of the grouped element

In general, we can use the `nth` function to get the *n*th row from a group, as illustrated:

```
grouped=df.groupby('Gender')
grouped.nth(1)
```

This gives the following result:



	Age	Height	Income	Socio-Eco	Weight
Gender					
Female	48.161761	83.708145	15242.128581	Middle Class	57.726991
Male	38.146687	168.017159	15990.565533	Middle Class	70.291151

First rows of each group

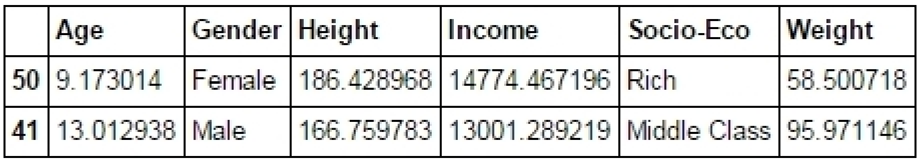
One can use any number (of course, less than the number of rows in each group) as the argument for the `nth` function.

It is always a good practice to sort the data frame for the relevant columns before creating the `groupby` object from the data frame. Suppose, you want to look at the youngest male and female members of this data frame.

This can be done by sorting the data frame, creating a `groupby` object, and then taking the first element of each group:

```
df1=df.sort(['Age','Income'])
grouped=df1.groupby('Gender')
grouped.head(1)
```

The output has two rows containing the details of the two youngest members from the two groups:



	Age	Gender	Height	Income	Socio-Eco	Weight
50	9.173014	Female	186.428968	14774.467196	Rich	58.500718
41	13.012938	Male	166.759783	13001.289219	Middle Class	95.971146

Sorting by the age column before grouping by gender and then selecting the first row from each group can give you the oldest/youngest guy in the group

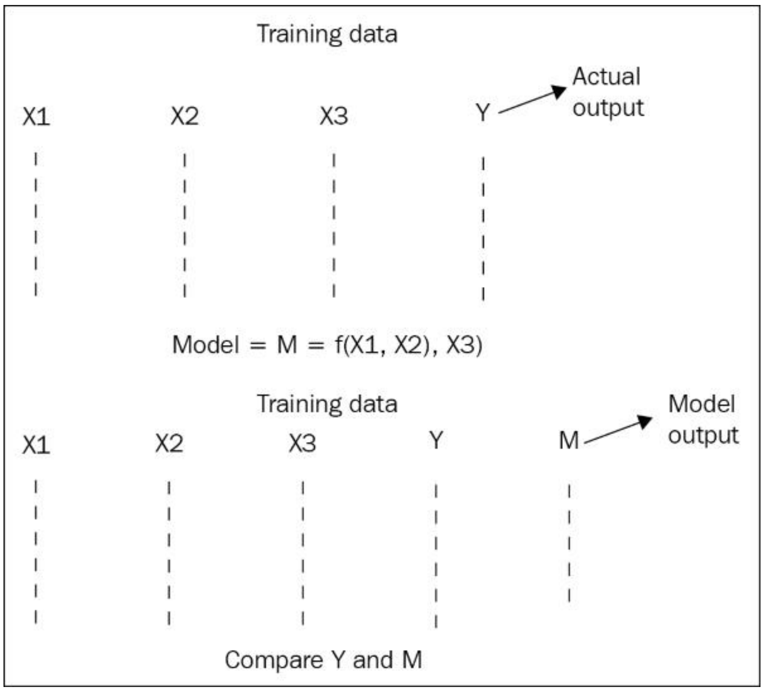
The oldest members can be identified in the same way by typing `grouped.tail(1)`.

Random sampling – splitting a dataset in training and testing datasets

Splitting the dataset in training and testing the datasets is one operation every predictive modeller has to perform before applying the model, irrespective of the kind of data in hand or the predictive model being applied. Generally, a dataset is split into training and testing datasets. The following is a description of the two types of datasets:

- The **training** dataset is the one on which the model is built. This is the one on which the calculations are performed and the model equations and parameters are created.
- The **testing** dataset is used to check the accuracy of the model. The model equations and parameters are used to calculate the output based on the inputs from the testing datasets. These outputs are used to compare the model efficiency in the light of the actuals present in the testing dataset.

This will become clearer from the following image:



Concept of sampling: Training and Testing data

Generally, the training and testing datasets are split in the ratio of 75:25 or 80:20. There are various ways to split the data into two halves. The crudest way that comes to mind is taking the first 75/80 percent rows as the training dataset and the rest as the testing dataset, or taking the first 25/20 percent rows as the testing and the rest as the training dataset. However, the problem with this approach is that it might bias the two datasets for a variety of reasons. The earlier rows might come from a different source or were observed during different scenarios. These situations might bias the model results from the two datasets. The rows should be chosen to avoid this bias. The most effective way to do that is to select the rows at random. Let us see a few methods to divide a dataset into training and testing datasets.

One way is to create as many standard normal random numbers, as there are rows in the dataset and then filter them for being smaller than a certain value. This filter condition is then used to partition the data in two parts. Let us see how it can be done.

Method 1 – using the Customer Churn Model

Let us use the same `Customer Churn Model` data that we have been using frequently. Let us go ahead and import it, as shown:

```
import pandas as pd
data = pd.read_csv('c:/FILE_PATH/Customer Churn Model.txt')
len(data)
```

There are 3333 rows in the dataset. Next, we will generate random numbers and create a filter on which to partition the data:

```
a=np.random.randn(len(data))
check=a<0.8
training=data[check]
testing=data[~check]
```

The rows where the value of the random number is less than 0.8 becomes a part of the training variable, while the one with a value greater than 0.8 becomes a part of the testing dataset.

Let us check the lengths of the two datasets to see in what ratio the dataset has been divided. A 75:25 split between training and testing datasets would be ideal:

```
len(training)
len(testing)
```

The length of training dataset is 2635 while that of the testing dataset is 698; thus, resulting in a split very close to 75:25.

Method 2 – using sklearn

Very soon we will be introduced to a very powerful Python library used extensively for the purpose of modelling, `scikit-learn` or `sklearn`. This `sklearn` library has inbuilt methods to split a dataset in a training and testing dataset. Let's have a look at the procedure:

```
from sklearn.cross_validation import train_test_split
train, test = train_test_split(data, test_size = 0.2)
```

The test size specifies the size of the testing dataset: 0.2 means that 20 percent of the rows of the dataset should go to testing and the remaining 80 percent to training. If we check the length of these two (train and test), we can confirm that the split is indeed 80-20 percent.

Method 3 – using the shuffle function

Another method involves using the `shuffle` function in the `random` method. The data is read in line by line, which are shuffled randomly and then assigned to training and testing datasets in designated proportions, as shown:

```
import numpy as np
with open('C:/FILE_PATH/Customer Churn Model.txt','rb') as f:
    data=f.read().split('\n')
np.random.shuffle(data)
train_data = data[:3*len(data)/4]
test_data = data[len(data)/4:]
```

In some cases, mostly during data science competitions like **Kaggle**, we would be provided with separate training and testing datasets to start with.

Concatenating and appending data

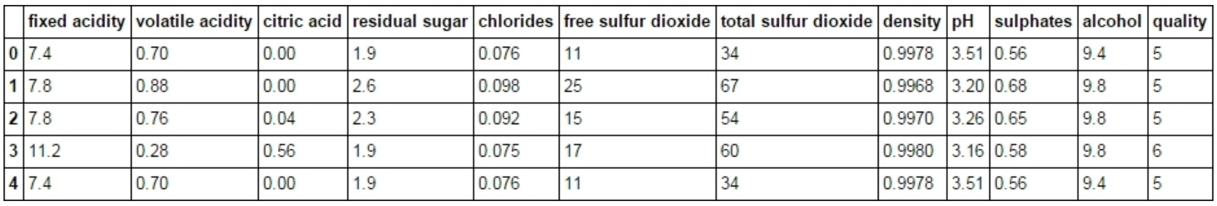
All the required information to build a model doesn't always come from a single table or data source. In many cases, two datasets need to be joined/merged to get more information (read new column/variable). Sometimes, small datasets need to be appended together to make a big dataset which contains the complete picture. Thus, merging and appending are important components of an analyst's armour.

Let's learn each of these methods one by one. For illustrating these methods, we will be using a lot of new interesting datasets. The one we are going to use first is a dataset about the mineral contents of wine; we will have separate datasets for red and white wine. Each sample represents a different sample of red or white wine.

Let us import this dataset and have a look at it. The delimiter for this dataset is `;` (a semi-colon), which needs to be taken care of:

```
import pandas as pd
data1=pd.read_csv('c:/FILE_PATH/winequality-red.csv',sep=';')
data1.head()
```

The output of this input snippet is similar to the following screenshot:

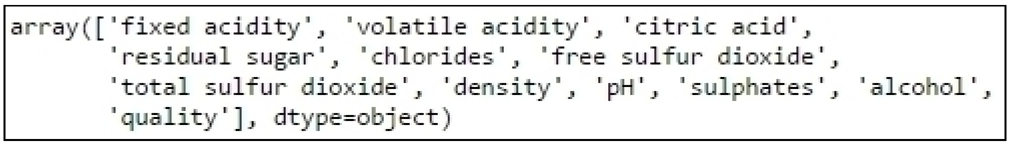


	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5

First few entries of the wine quality-red dataset

The column names are as follows:

```
data1.columns.values
```



```
array(['fixed acidity', 'volatile acidity', 'citric acid',
       'residual sugar', 'chlorides', 'free sulfur dioxide',
       'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol',
       'quality'], dtype=object)
```

Column names of the wine quality-red dataset

The size of the dataset is can be found out using the following snippet:

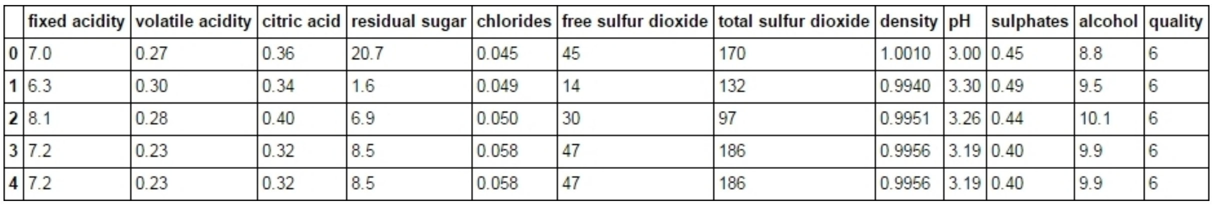
```
data1.shape
```

The output is 1599x12 implying that the dataset has 1599 rows.

Let us import the second dataset which is very similar to the preceding dataset except that the data points are collected for white wine:

```
import pandas as pd
data2=pd.read_csv(‘C:/FILE_PATH/Merge and Join/winequality-
white.csv',sep=';')
data2.head()
```

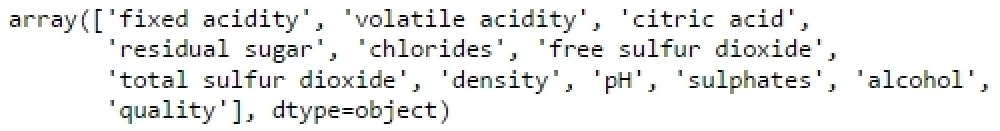
The output of this input snippet looks similar to the following screenshot:



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45	170	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14	132	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30	97	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47	186	0.9956	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47	186	0.9956	3.19	0.40	9.9	6

First few entries of the winequality-white dataset

As we can see, this dataset looks very similar to the preceding dataset. Let us confirm this by getting the column names for this dataset. They should be the same as the preceding array of column names:



```
array(['fixed acidity', 'volatile acidity', 'citric acid',
       'residual sugar', 'chlorides', 'free sulfur dioxide',
       'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol',
       'quality'], dtype=object)
```

Column names of the winequality-white dataset

The size of the dataset is, as follows:

```
data2.shape
```

4898x12, this means that the dataset has 4898 rows.

So, we can see that the `data1` and `data2` are very similar (in terms of column names and column types) except the row numbers in the two datasets. These are ideal circumstances to append two datasets along the horizontal axis (`axis=0`).

In Python, the horizontal axis is denoted by `axis=0` and the vertical axis is denoted by `axis=1`.

Let us append these two datasets along `axis=0`. This can be done using the `concat` method of `pandas` library. After appending the datasets, the row numbers of the final dataset should be the same as the row numbers of both the datasets.

This can be accomplished as follows:

```
wine_total=pd.concat([data1,data2],axis=0)
```

Let us check the number of rows of the appended dataset `wine_total`:

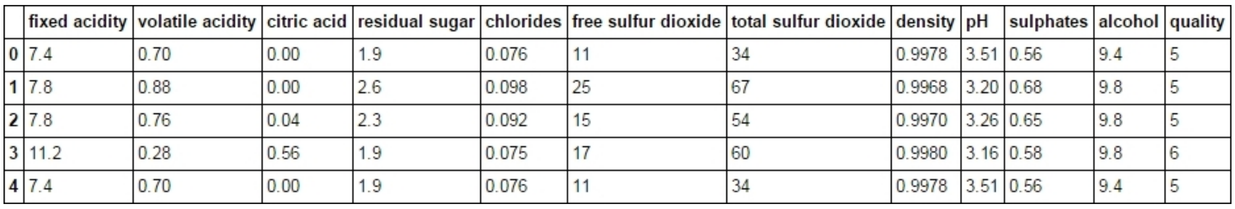
```
wine_total.shape
```

The output is `6497x12.` It indicates that the final appended dataset has 6497 (*6497=1599+4898*) rows. One can see that the row numbers in the appended dataset is the sum of row numbers of the individual datasets.

Let us have a look at the final dataset just to ensure everything looks fine. While appending over `axis=0`, the two datasets are just stacked over one another. In this case, `data1` will be stacked over `data2` dataset. So, the first few rows of the final dataset `wine_total` will look similar to the first few rows of the first dataset `data1`. Let us check that:

```
wine_total.head()
```

The output looks similar to the following screenshot:



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5

First few entries of the final dataset obtained from appending data1 and data2

The preceding output is the same as the first few rows of the `data1`.

This `concat` method can be used to scramble data that is taken a few rows from here and there and stacking them over one another. The `concat` method takes more than two datasets also as an argument. The datasets are stacked over one another in order of appearance. If the datasets are `data1`, `data2`, and `data3`, in that order, then `data1` will be stacked over `data2` which will be stacked over `data3`.

Let us look at an example of such scrambling. We will use the `data1` dataset (coming from `winequality-red.csv`) and take 50 rows from head, middle, and tail to create three different data frames. These data frames will be then stacked over one another to create a final dataset:

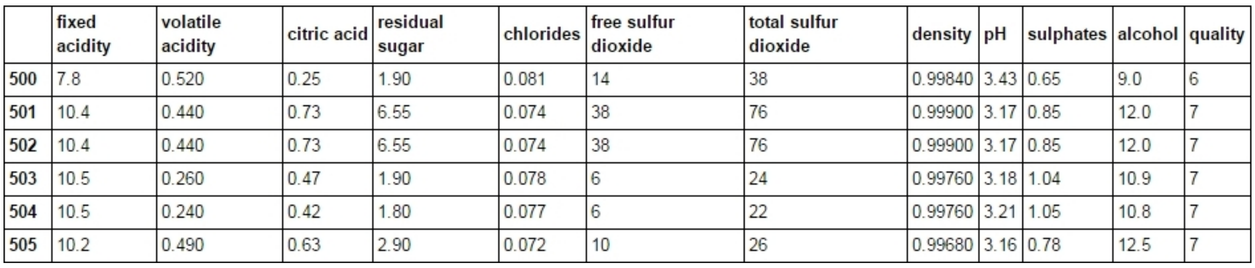
```
data1_head=data1.head(50)
data1_middle=data1[500:550]
data1_tail=data.tail(50)
wine_scramble=pd.concat([data1_middle,data1_head,data1_tail],axis=0)
wine_scramble
```

The output dataset will contain 150 rows, as confirmed by the following snippet:

```
wine_scramble.shape
```

This returns 150x12 as the output.

The output dataset `wine_scramble` looks similar to the following screenshot:



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
500	7.8	0.520	0.25	1.90	0.081	14	38	0.99840	3.43	0.65	9.0	6
501	10.4	0.440	0.73	6.55	0.074	38	76	0.99900	3.17	0.85	12.0	7
502	10.4	0.440	0.73	6.55	0.074	38	76	0.99900	3.17	0.85	12.0	7
503	10.5	0.260	0.47	1.90	0.078	6	24	0.99760	3.18	1.04	10.9	7
504	10.5	0.240	0.42	1.80	0.077	6	22	0.99760	3.21	1.05	10.8	7
505	10.2	0.490	0.63	2.90	0.072	10	26	0.99680	3.16	0.78	12.5	7

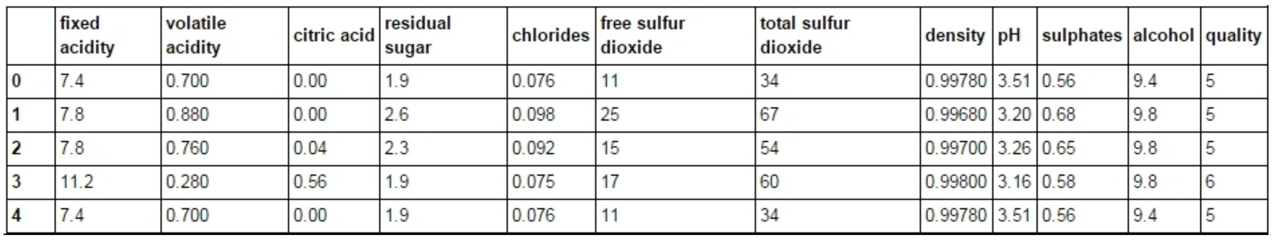
First few rows of the scrambled data frame with rows from the data1_middle at the top

Since, the order of the appended dataset is `data1_middle`, `data1_head`, `data1_tail`, the rows contained in the `data1_middle` come at the top followed by the `data1_head` and `data1_tail` rows.

If you change the order of the stacking, the view of the appended dataset will change. Let's try that:

```
data1_head=data1.head(50)
data1_middle=data1[500:550]
data1_tail=data.tail(50)
wine_scramble=pd.concat([data1_head,data1_middle,data1_tail],axis=0)
wine_scramble
```

The output looks similar to the following screenshot, wherein, as expected the rows in the `data1_head` appear at the top:



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11	34	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25	67	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15	54	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17	60	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11	34	0.99780	3.51	0.56	9.4	5

First few rows of the scrambled data frame with rows from the data1_head at the top

Let's see another scenario where the `concat` function comes as a saviour. Suppose, you have to deal with the kind of data that comes in several files containing similar data. One example of such a scenario is the daily weather data of a city where the same metrics are tracked everyday but the data for each day is stored in a separate file. Many a time, when we do analysis of such files, we are required to append them into a single consolidated file before running any analyses or models.

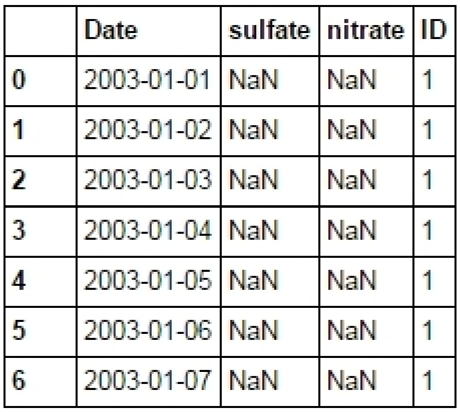
I have curated some data that has these kind of properties to illustrate the method of consolidation into a single file. Navigate to the `lab` folder and then to the `lotofdata` folder. You will see 332 CSV files, each named with its serial number. Each CSV file contains pollutant measure levels at different points of time in a single day. Each CSV file represents a day worth of pollutant measure data.

Let us go ahead and import the first file and have a look at it. Looking at one file will be enough, as the others will be very similar to this (exactly similar except the number of rows and the data points):

```
data=pd.read_csv('C:/FILE_PATH Datasets/Merge and Join/lotofdata/001.csv')
data.head()
```

The one feature of this data is that it is very sparse and it contains a lot of missing values that would be visible once we look at the output of the preceding code snippet. The sparseness doesn't affect the analyses in this case because the dataset has sufficient rows with non-missing values. Even 100 rows with non-missing values should give us a good picture of the pollutant levels for a day. However, for the same reason, appending such a dataset becomes all the more important so that we have a significant amount of data with the non-missing values for our analyses.

Let us look at the first CSV file of the lot and the output of the preceding snippet:



	Date	sulfate	nitrate	ID
0	2003-01-01	NaN	NaN	1
1	2003-01-02	NaN	NaN	1
2	2003-01-03	NaN	NaN	1
3	2003-01-04	NaN	NaN	1
4	2003-01-05	NaN	NaN	1
5	2003-01-06	NaN	NaN	1
6	2003-01-07	NaN	NaN	1

First few entries of the first file out of 332 CSV files

The size of the dataset is 1461x4 indicating that there are 1,461 rows and four columns. The name of the columns are **Date**, **sulfate**, **nitrate**, and **ID**. **ID** would be **1** for the first dataset, **2** for the 2nd, and so on. The number of rows in the other CSV files should be in the same range while the number of rows with non-missing values might vary.

Let us now move towards our goal of this discussion that is to demonstrate how to consolidate such small and similar files in a single file. To be able to do so, one needs to do the following in that sequence:

1. Import the first file.
2. Loop through all the files.
3. Import them one by one.
4. Append them to the first file.

5. Repeat the loop.

Let us now look at the code snippet, which will achieve this:

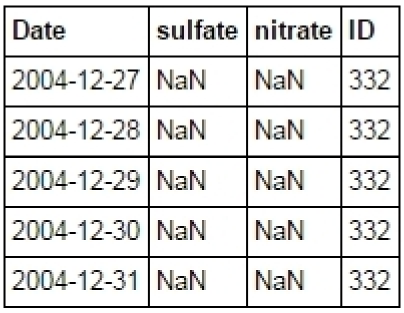
```
import pandas as pd
filepath=‘C:/FILE_PATH/Merge and Join/lotofdata'
data_final=pd.read_csv('C:/FILE_PATH/Merge and Join/lotofdata/001.csv')
for i in range(1,333):
    if i<10:
        filename='0'+'0'+str(i)+'.csv'
    if 10<=i<100:
        filename='0'+str(i)+'.csv'
    if i>=100:
        filename=str(i)+'.csv'

    file=filepath+'/'+filename
    data=pd.read_csv(file)

    data_final=pd.concat([data_final,data],axis=0)
```

In the code snippet, the `read_csv` is taking a file variable that consists of `filepath` and `filename` variables. The `if` condition takes care of the changing filenames (three conditions arise – first, when filename contains a single non-zero digit; second, when the filename contains two non-zero digits, and third, when the filename contains all the three non-zero digits).

The first file is imported and named as `data_final`. The subsequent files are imported and appended to `data_final`. The `for` loop runs over all the 332 files wherein the importing and appending of the files occur.

The size of the `data_final` data frame is 773548x4 rows, indicating that it has 7,73,548 rows because it contains the rows from all the 332 files.

If one looks at the last rows of the `data_final` data frame, one can confirm that all the files have been appended if the **ID** column contains 332 as value. This means that the last few rows come from the 332nd file.



Date	sulfate	nitrate	ID
2004-12-27	NaN	NaN	332
2004-12-28	NaN	NaN	332
2004-12-29	NaN	NaN	332
2004-12-30	NaN	NaN	332
2004-12-31	NaN	NaN	332

Last few entries of the data_final data frame. They have ID as 332 indicating that they come from the 332nd CSV file.

The **ID** column indeed contains 332 observations, confirming that all the 332 files have been successfully appended.

Another way to confirm whether all the rows from all the files have been successfully appended or not, one can sum up the row numbers of each file and compare them to the row numbers of the final appended data frame. They should be equal if they have been appended successfully.

Let us check. We can use the same code as the preceding one, for this process with some minor tweaks. Let us see how:

```
import pandas as pd
filepath=‘C:/FILE_PATH/Merge and Join/lotofdata'
data_final=pd.read_csv(‘C:/FILE_PATH/Merge and Join/lotofdata/001.csv')
data_final_size=len(data_final)
for i in range(1,333):
    if i<10:
        filename='0'+'0'+str(i)+'.csv'
    if 10<=i<100:
        filename='0'+str(i)+'.csv'
    if i>=100:
        filename=str(i)+'.csv'

    file=filepath+'/'+filename
    data=pd.read_csv(file)
    data_final_size+=len(data)
    data_final=pd.concat([data_final,data],axis=0)
print data_final_size
```

Here, we are summing-up the row numbers of all the files (in the line highlighted) and the summed-up number is printed in the last line. The output is 773,548; it confirms that the final data frame has the same number of rows as the sum of rows in all the files.

Merging/joining datasets

Merging or joining is a mission critical step for predictive modelling and, more often than not, while working on actual problems, an analyst will be required to do it. Similar to relational databases where there are multiple tables connected by a common key column across which the required columns are scattered. There can be instances where two tables are joined by more than one key column. The merges and joins in Python are very similar to a table merge/join in a relational database except that it doesn't happen in a database but rather on the local computer and that these are not tables, rather data frames in pandas. For people familiar with Excel, you can find similarity with the `VLOOKUP` function in the sense that both are used to get an extra column of information from a sheet/table joined by a key column.

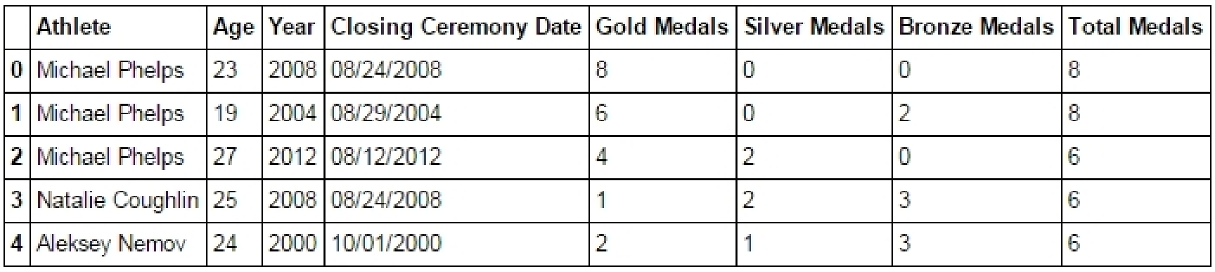
There are various ways in which two tables/data frames can be merged/joined. The most commonly used ones are Inner Join, Left Join, Right Join, and so on. We will go in to detail and understand what each of these mean. But before that, let's go ahead and perform a simple merge to get a feel of how it is done.

We will be using a different dataset to illustrate the concept of merge and join. These datasets can be found in the lab folder in `Merge` and the `Join/Medals` folder. The main dataset `Medals.csv` contains details of medals won by individual players at different Olympic events. The two subsidiary datasets contain details of the nationality and sports of the individual player. What if we want to see the nationality or sport played by the player together with all the other medal information for each player? The answer is to merge both the datasets and to get the relevant columns. In data science parlance, merging, joining, and mapping are used synonymously; although, there are minor technical differences.

Let us import all of them and have a cursory look at them:

```
import pandas as pd
data_main=pd.read_csv(‘C:/FILE_PATH/Merge and Join/Medals/Medals.csv')
data_main.head()
```

The `Medals.csv` looks similar to the following screenshot:



	Athlete	Age	Year	Closing Ceremony Date	Gold Medals	Silver Medals	Bronze Medals	Total Medals
0	Michael Phelps	23	2008	08/24/2008	8	0	0	8
1	Michael Phelps	19	2004	08/29/2004	6	0	2	8
2	Michael Phelps	27	2012	08/12/2012	4	2	0	6
3	Natalie Coughlin	25	2008	08/24/2008	1	2	3	6
4	Aleksey Nemov	24	2000	10/01/2000	2	1	3	6

First few entries of the Medals dataset

As we can see, this is the information about the **Olympic Year** in which the medals were won, details of how many Gold, Silver, and Bronze medals were won and the **Age** of the player. There are 8,618 rows in the dataset. One more thing one might be interested to know about this dataset is how many unique athletes are there in the dataset, which will come in handy later when we learn and apply different kinds of joins:

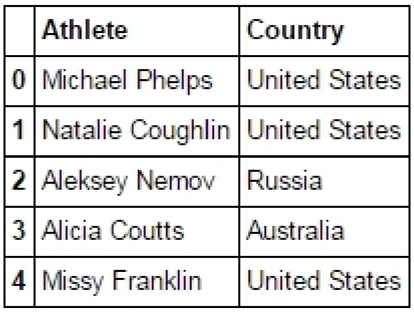
```
a=data_main['Athlete'].unique().tolist()
len(a)
```

The output of this snippet is `6956`, which means that there are many athletes for whom we have records in the datasets. The other entries come because many athletes may have participated in more than one Olympics.

Let us now import the `Athelete Country Map.csv` and have a look at it:

```
country_map=pd.read_csv('C:/FILE_PATH/Merge and
Join/Medals/Athelete_Country_Map.csv')
country_map.head()
```

The output data frame looks similar to the following screenshot, with two columns: **Athlete** and **Country**:



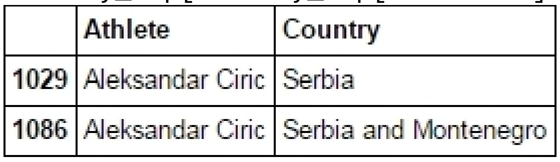
	Athlete	Country
0	Michael Phelps	United States
1	Natalie Coughlin	United States
2	Aleksey Nemov	Russia
3	Alicia Coutts	Australia
4	Missy Franklin	United States

First few entries of the Athelete_Country_Map dataset

There are 6,970 rows in this dataset. If you try to find out the unique number of athletes in this data frame, it will still be 6,956. The 14 extra rows come from the fact that some players have played for two countries in different Olympics and have won medals. Search for `Aleksandar Ciric` and you will find that he has played for both `Serbia` and `Serbia and Montenegro`.

You can do this by using the following code snippet:

```
country_map[country_map['Athlete']=='Aleksandar Ciric']
```



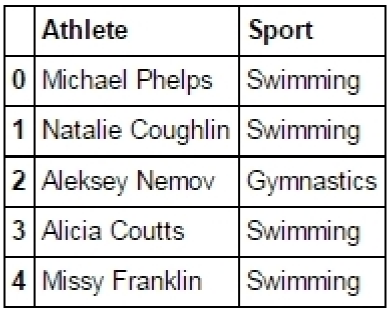
	Athlete	Country
1029	Aleksandar Ciric	Serbia
1086	Aleksandar Ciric	Serbia and Montenegro

Subsetting the country_map data frame for Aleksandar Ciric

Let us finally import the `Athelete Sports Map.csv` and have a look at it:

```
sports_map=pd.read_csv('C:/FILE_PATH/Merge and
Join/Medals/Athelete_Sports_Map.csv')
sports_map.head()
```

The `sports_map` data frame looks as shown in the following screenshot:



	Athlete	Sport
0	Michael Phelps	Swimming
1	Natalie Coughlin	Swimming
2	Aleksey Nemov	Gymnastics
3	Alicia Coutts	Swimming
4	Missy Franklin	Swimming

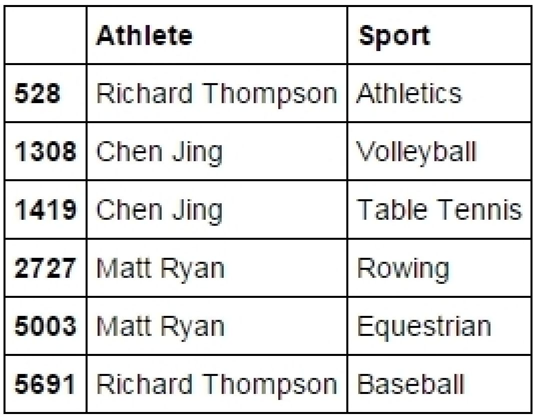
First few entries of the Athelete_Sports_Map dataset

There are 6,975 rows in this dataset because, yes you guessed it right, there are very few athletes in this mapping data frame who have played more than one game and have won medals. Watch out for athletes, such as `Chen Jing`, `Richard Thompson` and `Matt Ryan` who have played more than one game.

This can be done by writing a code, such as the following snippet:

```
sports_map[(sports_map['Athlete']=='Chen Jing') |
(sports_map['Athlete']=='Richard Thompson') | (sports_map['Athlete']=='Matt
Ryan')]
```

The output looks similar to the following screenshot:



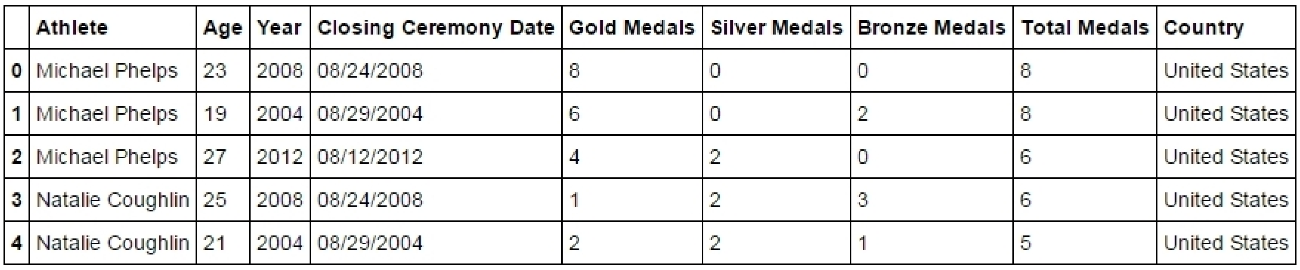
	Athlete	Sport
528	Richard Thompson	Athletics
1308	Chen Jing	Volleyball
1419	Chen Jing	Table Tennis
2727	Matt Ryan	Rowing
5003	Matt Ryan	Equestrian
5691	Richard Thompson	Baseball

Subsetting the sports_map data frame for athletes Richard Thompson and Matt Ryan

Let's now merge the `data_main` and `country_map` data frames to get the country for all the athletes. There is a `merge` method in pandas, which facilitates this:

```
import pandas as pd
merged=pd.merge(left=data_main,right=country_map,left_on='Athlete',right_on
='Athlete')
merged.head()
```

The output looks, as follows. It has a country column as expected:

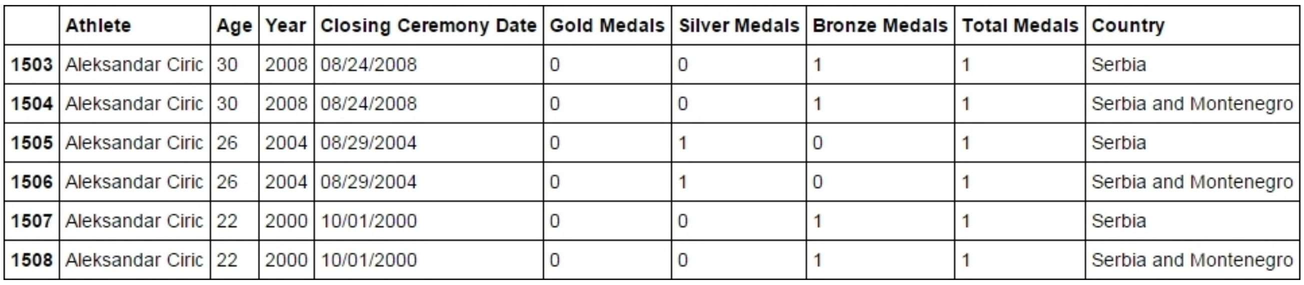


	Athlete	Age	Year	Closing Ceremony Date	Gold Medals	Silver Medals	Bronze Medals	Total Medals	Country
0	Michael Phelps	23	2008	08/24/2008	8	0	0	8	United States
1	Michael Phelps	19	2004	08/29/2004	6	0	2	8	United States
2	Michael Phelps	27	2012	08/12/2012	4	2	0	6	United States
3	Natalie Coughlin	25	2008	08/24/2008	1	2	3	6	United States
4	Natalie Coughlin	21	2004	08/29/2004	2	2	1	5	United States

First few entries of the merged data frame. It has a country column.

The length of the merged data frame is 8,657, which is more than the total number of rows (8,618) in the `data_main` data frame. This is because when we join these two data frames without any specified conditions, an inner join is performed wherein the join happens based on the common key-values present in both the data frames. Also, we saw that some athletes have played for two countries and the entries for such athletes will be duplicated for such athletes. If you look at `Aleksandar Ciric` in the merged data frame, you will find something similar to this:

```
merged[merged['Athlete']=='Aleksandar Ciric']
```



	Athlete	Age	Year	Closing Ceremony Date	Gold Medals	Silver Medals	Bronze Medals	Total Medals	Country
1503	Aleksandar Ciric	30	2008	08/24/2008	0	0	1	1	Serbia
1504	Aleksandar Ciric	30	2008	08/24/2008	0	0	1	1	Serbia and Montenegro
1505	Aleksandar Ciric	26	2004	08/29/2004	0	1	0	1	Serbia
1506	Aleksandar Ciric	26	2004	08/29/2004	0	1	0	1	Serbia and Montenegro
1507	Aleksandar Ciric	22	2000	10/01/2000	0	0	1	1	Serbia
1508	Aleksandar Ciric	22	2000	10/01/2000	0	0	1	1	Serbia and Montenegro

Subsetting the merged data frame for athlete Aleksandar Ciric

The problem is not with the type of join but with the kind of mapping file we have. This mapping file is one-many and hence the number increases because for each key multiple rows are created in such a case.

To rectify this issue, one can remove the duplicate entries from the `country_map` data frame and then perform the merge with `data_main`. Let's do that. This can be done using the `drop_duplicates` method, as shown:

```
country_map_dp=country_map.drop_duplicates(subset='Athlete')
```

The length of the `country_map_dp` is 6,956 rows, which is the same as the number of unique athletes. Let us now merge this with `data_main`.

```
merged_dp=pd.merge(left=data_main,right=country_map_dp,left_on='Athlete',ri
ght_on='Athlete')
len(merged_dp)
```

The number of rows in the `merged_dp` is indeed 8,618, which is the actual number of rows in the `data_main`.

The next step is to merge `sports_map` with the `merged_dp` to get the country and sports along with other details in the same data frame.

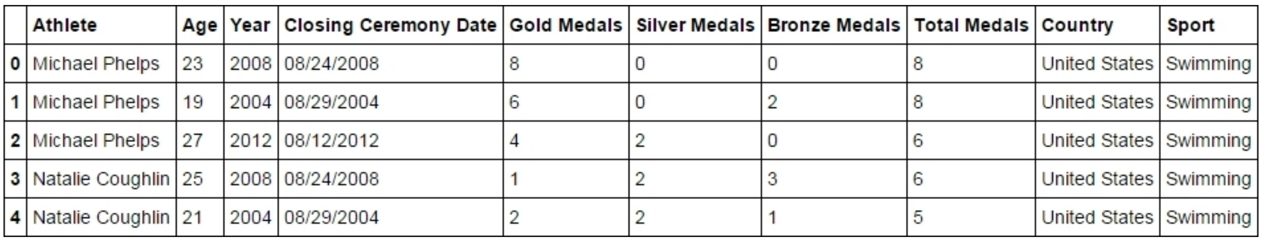
We have seen similar issue of increase in the number of rows for `sports_map`, as was the case for `country_map` data frame. To take care of that, let's remove the duplicates from the `sports_map` before merging it with `merged_dp`:

```
sports_map_dp=sports_map.drop_duplicates(subset='Athlete')
len(sports_map_dp)
```

The length of the `sports_map_dp` is 6,956, which is the same as the number of rows in the `data_main` data frame, as expected.

The next step is to merge this with the `merge_pd` data frame to get the sports played by the athlete in the final merged table:

```
merged_final=pd.merge(left=merged_dp,right=sports_map_dp,left_on='Athlete',
right_on='Athlete')
merged_final.head()
```



	Athlete	Age	Year	Closing Ceremony Date	Gold Medals	Silver Medals	Bronze Medals	Total Medals	Country	Sport
0	Michael Phelps	23	2008	08/24/2008	8	0	0	8	United States	Swimming
1	Michael Phelps	19	2004	08/29/2004	6	0	2	8	United States	Swimming
2	Michael Phelps	27	2012	08/12/2012	4	2	0	6	United States	Swimming
3	Natalie Coughlin	25	2008	08/24/2008	1	2	3	6	United States	Swimming
4	Natalie Coughlin	21	2004	08/29/2004	2	2	1	5	United States	Swimming

First few entries of the merged_final dataset. The duplicates from country_map were deleted before the merge

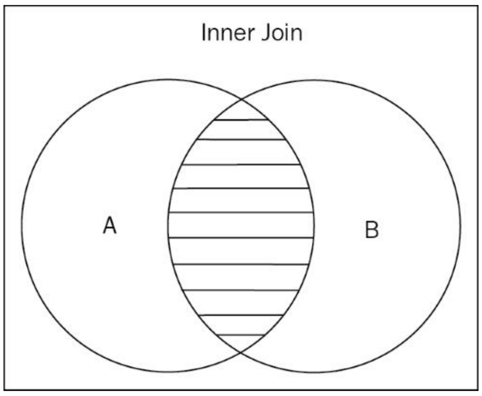
As we can see, the **Sport** column is present in the `merged_final` data frame after the merge. The `merged_final` data frame has 8,618 rows as expected.

Let us now look at various kinds of merge/joins that we can apply to two data frames. Although you would come across many kinds of joins in different texts, it is sufficient to know the concept behind the three of them—Inner Join, Left Join, and Right Join. If you consider the two tables/data frames as sets, then these joins can be well represented by **Venn Diagrams**.

Inner Join

The characteristics of the Inner Join are as follows:

- Returns a data frame containing rows, which have a matching value in both the original data frames being merged.
- The number of rows will be equal to the minimum of the row numbers of the two data frames. If data frame *A* containing 100 rows is being merged with data frame *B* having 80 rows, the merged data frame will have 80 rows.
- The Inner Join can be thought of as an intersection of two sets, as illustrated in the following figure:

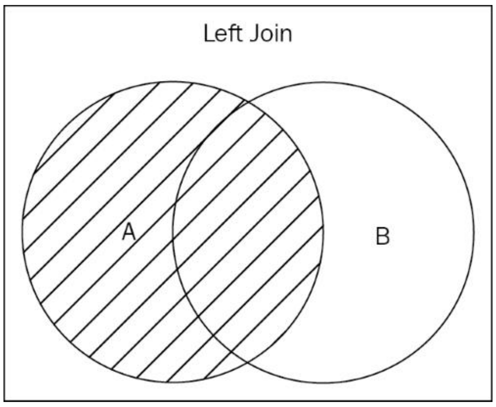


Inner Join illustrated via a Venn diagram

Left Join

The characteristics of the Left Join are, as follows:

- Returns a data frame containing rows, which contains all the rows from the left data frame irrespective of whether it has a match in the right data frame or not.
- In the final data frame, the rows with no matches in the right data frame will return NAs in the columns coming from right data frame.
- The number of rows will be equal to the number of rows in the left data frame. If data frame *A* containing 100 rows is being merged with data frame *B* having 80 rows, the merged data frame would have 100 rows.
- The Left Join can be thought of as the set containing the entire left data frame, as illustrated in the following figure:

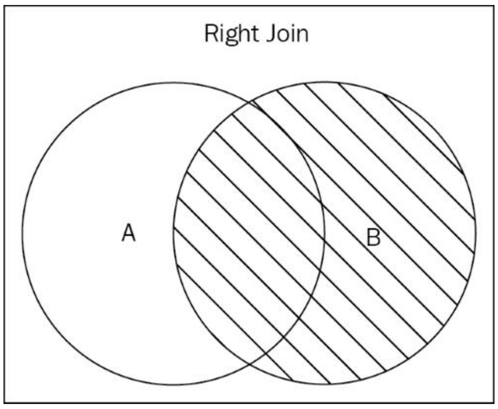


Left Join illustrated via a Venn Diagram

Right Join

The characteristics of the Right Join are as follows:

- Returns a data frame containing rows, which contains all the rows from the right data frame irrespective of whether it has a match in the left data frame or not.
- In the final data frame, the rows with no matches in the left data frame will return NAs in the columns coming from left data frame.
- The number of rows will be equal to the number of rows in the left data frame. If data frame *A* containing 100 rows is being merged with data frame *B* having 80 rows, the merged data frame will have 80 rows
- The Right Join can be thought of as the set containing the entire right data frame, as illustrated in the following figure:



Right Join illustrated via a Venn diagram

The comparison between join type and set operation is summarized in the following table:

Join type	Set operation
Inner Join	Intersection
Left Join	Set **A** (left data frame)
Right Join	Set **B** (right data frame)
Outer Join	Union

Let us see some examples of how different kinds of mappings actually work. For that, a little data preparation is needed. Currently, both our mapping files contain matching entries for all the rows in the actual data frame `data_main`. So, we can't see the effects of different kind of merges. Let's create a country and sports mapping file which doesn't have the information for some of the athletes and let's see how it reflects in the merged table. This can be done by creating a new data frame that doesn't have country/sports information for some of the athletes, as shown in the following code:

```
country_map_dlt=country_map_dp[(country_map_dp['Athlete']<>'Michael
Phelps') & (country_map_dp['Athlete']<>'Natalie Coughlin') &
(country_map_dp['Athlete']<>'Chen Jing')
                    & (country_map_dp['Athlete']<>'Richard Thompson') &
(country_map_dp['Athlete']<>'Matt Ryan')]
len(country_map_dlt)
```

Using this snippet, we have created a `country_map_dlt` data frame that doesn't have country mapping for five athletes, that is `Michael Phelps`, `Natalie Coughlin`, `Chen Jing`, `Richard Thompson`, and `Matt Ryan`. The length of this data frame is 6,951; it is five less than the actual mapping file, indicating that the information for five athletes has been removed.

Let's do the same for `sports_map` as well as the `data_main` data frame using the following snippets:

```
sports_map_dlt=sports_map_dp[(sports_map_dp['Athlete']<>'Michael Phelps') &
(sports_map_dp['Athlete']<>'Natalie Coughlin') &
(sports_map_dp['Athlete']<>'Chen Jing')
                    & (sports_map_dp['Athlete']<>'Richard Thompson') &
(sports_map_dp['Athlete']<>'Matt Ryan')]
len(sports_map_dlt)

data_main_dlt=data_main[(data_main['Athlete']<>'Michael Phelps') &
(data_main['Athlete']<>'Natalie Coughlin') & (data_main['Athlete']<>'Chen
Jing')
                    & (data_main['Athlete']<>'Richard Thompson') &
(data_main['Athlete']<>'Matt Ryan')]
len(data_main_dlt)
```

The length of `data_main_dlt` becomes 8,605 because the `data_main` contains multiple rows for an athlete.

An example of the Inner Join

One example of Inner join would be to merge `data_main` data frame with `country_map_dlt`. This can be done using the following snippet:

```
merged_inner=pd.merge(left=data_main,right=country_map_dlt,how='inner',left
_on='Athlete',right_on='Athlete')
len(merged_inner)
```

This merge should give us information for the athletes who are present in both the data frames. As the `country_map_dlt` doesn't contain information about five athletes present in `data_main`, these five athletes wouldn't be a part of the merged table.

The length of the `merged_inner` comes out to be 8,605 (similar to `data_main_dlt`) indicating that it doesn't contain information about those five athletes.

An example of the Left Join

One example of Left Join would be to merge `data_main` data frame with `country_map_dlt`. This can be done using the following snippet:

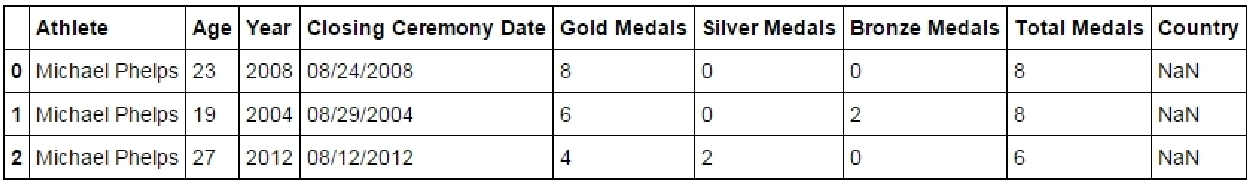
```
merged_left=pd.merge(left=data_main,right=country_map_dlt,how='left',left_o
n='Athlete',right_on='Athlete')
len(merged_left)
```

This merge should give us the information about all the athletes that are present in the left data frame (`data_main`) even if they aren't present in the right data frame (`country_map_dlt`). So, the `merged_left` data frame should contain 8,618 rows (similar to the `data_main`) even if the `country_map_dlt` doesn't contain information about five athletes present in `data_main`. These five athletes will have a **NaN** value in the **Country** column.

The length of `merged_left` indeed comes out to be 8,618. Let's check the `merged_left` for an athlete whose information is not present in the `country_map_dlt`. It should contain **NaN** for the **Country** column:

```
merged_left_slt=merged_left[merged_left['Athlete']=='Michael Phelps']
merged_left_slt
```

The output is similar to the following screenshot. It indeed contains **NaN** for **Michael Phelps**' **Country** because it doesn't have a mapping in `country_map_dlt`:



	Athlete	Age	Year	Closing Ceremony Date	Gold Medals	Silver Medals	Bronze Medals	Total Medals	Country
0	Michael Phelps	23	2008	08/24/2008	8	0	0	8	NaN
1	Michael Phelps	19	2004	08/29/2004	6	0	2	8	NaN
2	Michael Phelps	27	2012	08/12/2012	4	2	0	6	NaN

Merged_left data frame sub-setted for Michael Phelps contains NaN values, as expected

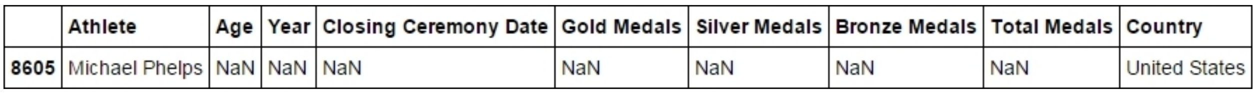
An example of the Right Join

One example of Right Join will be to merge data frame `data_main` with `country_map_dlt.` This can be done using the following snippet:

```
merged_right=pd.merge(left=data_main_dlt,right=country_map_dp,how='right',l
eft_on='Athlete',right_on='Athlete')
len(merged_right)
```

This should contain the **NaN** values for the columns coming from `data_main_dlt`, in the rows where there is no athlete information in `data_main_dlt`.

As shown in the following table:



	Athlete	Age	Year	Closing Ceremony Date	Gold Medals	Silver Medals	Bronze Medals	Total Medals	Country
8605	Michael Phelps	NaN	NaN	NaN	NaN	NaN	NaN	NaN	United States

merged_right data frame sub-setted for Michael Phelps contains NaN values, as expected

There will be one row created for each athlete who is not there in the `data_main_dlt` but is present in the `country_map_dp`. Hence, there will be five extra rows, one for each deleted athlete. The number of rows in the `merged_right` is thus equal to 8,610.

There are other joins like Outer Joins, which can be illustrated as the Union of two data frames. The Outer join would contain rows from both the data frames, even if they are not present in the other. It will contain `NaN` for the columns which it can't get values for. It can be easily performed setting the `how` parameter of the `merge` method to `outer`.

Summary of Joins in terms of their length

The effect of these joins can be more effectively explained if we summarize the number of samples present in the data frames that were used for merging and in the resultant data frames.

The first table provides the number of samples present in the data frames that were used for merging. All these data frames have been defined earlier in this section of the lab:

Data frame	Length (# rows)
`data_main`	8618
`data_main_dlt`	8605
`country_map_dp`	6956
`country_map_dlt`	6951

This table provides the number of samples present in the merged data frames:

Merged data frame	Components	Length
`merged_inner`	`data_main` with `country_map_dlt`	8605
`merged_left`	`data_main` with `country_map_dlt`	8618
`merged_right`	`data_main_dlt` with `country_ma_dp`	8610

Summary

Quite a long lab! Isn't it? But, this lab will form the core of anything you learn and implement in data-analysis. Let us wrap-up the lab by summarizing the key takeaways:

- Data can be sub-setted in a variety of ways: by selecting a column, selecting few rows, selecting a combination of rows and columns; using `.ix` method and `[ ]` method, and creating new columns.
- Random numbers can be generated in a number of ways. There are many methods like `randint()`, `raandarrange()` in the `random` library of `numpy`. There are also methods like `shuffle` and `choice` to randomly select an element out of a list. `Randn()` and `uniform()` are used to generate random numbers following normal and uniform probability distributions. Random numbers can be used to run simulations and generate dummy data frames.
- The `groupby()` method creates a `groupby` element on which `aggregate`, `transform`, and `filter` operations can be applied. This is a good method to summarize data for each categorical variable at once.
- A data must be split between training and testing datasets before a modelling is performed. The training dataset is the one on which the model equations are developed. The testing dataset is used to test the performance of the model comparing the actual result (present in testing dataset) to the model output. There are various ways to perform this split. One can use `choice` and `shuffle`. Scikit-learn has a readymade method for this.
- Two datasets can be merged just like two tables in a relational database. There are various kind of joins—Inner, Left, Right, Outer, and so on. These joins can be understood better if the datasets are assumed analogous to sets. Inner Join is then Intersection, Outer Join is Union, and Left and Right joins are entire left and right data frame.

Pre-processing data and bringing it in the form you desire is a big challenge before one proceeds to modelling. But, once done, it opens up a plethora of insights and information to be discovered using predictive models.