In this module we are going to learn to handle datasets; creating datasets, reading from external files , modifying datasets. We will also see summarizing and visualizing in python with packages numpy, pandas and seaborn.

Numpy

Through numpy we will learn to create and handle arrays. Arrays set a background for handling columns in our datasets when we eventually move to pandas dataframes.
We will cover the following topics in Numpy:

```
a. creating nd arrays

b. subsetting with indices and conditions

c. comparison with np and math functions [np.sqrt , log etc ] and special numpy functions
```

For this course, we will consider only two dimensional arrays, though technically, we can create arrays with more than two dimensions.

We start with importing the package numpy giving it the alias np.

```
import numpy as np
```

We start with creating a 2 dimensional array and assign it to the variable 'b'. It is simply a list of lists.

```
b = np.array([[3,20,99],[-13,4.5,26],[0,-1,20],[5,78,-19]])
b
```

```
array([[  3. ,  20. ,  99. ],
       [-13. ,   4.5,  26. ],
       [  0. ,  -1. ,  20. ],
       [  5. ,  78. , -19. ]])
```

We have passed 4 lists and each of the lists contains 3 elements. This makes 'b' a 2 dimensional array.

We can also determine the size of an array by using its shape attribute.

```
b.shape
```

```
(4, 3)
```

Each dimension in a numpy array is referred to by the argument 'axis'. 2 dimensional arrays have two axes, namely 0 and 1. Since there are two axes here, we will need to pass two indices when accessing the values in the numpy array. Numbering along both the axes starts with a 0.

```
b
```

```
array([[  3. ,  20. ,  99. ],
       [-13. ,   4.5,  26. ],
       [  0. ,  -1. ,  20. ],
       [  5. ,  78. , -19. ]])
```

Assuming that we want to access the value -1 from the array 'b', we will need to access it with both its indices along the two axes.

```
b[2,1]
```

```
-1.0
```

Subset an array using Indexing and Slicing

In order to access -1, we will need to first pass index 2 which refers to the third list and then we will need to pass index 1 which refers to the second element in the third list. Axis 0 as refers to each list and axis 1 refers to elements present in each list. First index is the index of the list where the element is (i.e. 2) and the second index is its position within that list (i.e. 1).

Indexing and slicing here works just like it did in lists, only difference being that here we are considering 2 dimensions.

Now lets consider a few more examples

```
print(b)
b[:,1]
```

```
[[  3.   20.   99. ]
 [-13.    4.5  26. ]
 [  0.   -1.   20. ]
 [  5.   78.  -19. ]]
```

```
array([20. ,  4.5, -1. , 78. ])
```

This statement gives us the second element from all the lists.

```
b[1,:]
```

```
array([-13. ,   4.5,  26. ])
```

The above statement gives us all the elements from the second list.

By default, we can access all the elements of a list by providing a single index as well. The above code can also be written as:

```
b[1]
```

```
array([-13. ,   4.5,  26. ])
```

We can access multiple elements of a 2 dimensional array by passing multiple indices as well.

```
print(b)
b[[0,1,1],[1,2,1]]
```

```
[[  3.   20.   99. ]
 [-13.    4.5  26. ]
 [  0.   -1.   20. ]
 [  5.   78.  -19. ]]
```

```
array([20. , 26. ,  4.5])
```

Here, values are returned by pairing of indices i.e. (0,1), (1,2) and (1,1). We will get the first element when we run b[0,1] (i.e. the first list and second element within that list); the second element when we run b[1,2] (i.e. the second list and the third element within that list) and the third element when we run b[1,1] (i.e. the second list and the second element within that list). The three values returned in the array above can also be obtained by the three print statements written below:

```
print(b[0,1])
print(b[1,2])
print(b[1,1])
```

```
20.0
26.0
4.5
```

This way of accessing the index can be used for modification as well e.g. updating the values of those indices.

```
print(b)
b[[0,1,1],[1,2,1]]=[-10,-20,-30]
print(b)
```

```
[[  3.   20.   99. ]
 [-13.    4.5  26. ]
 [  0.   -1.   20. ]
 [  5.   78.  -19. ]]
[[  3. -10.  99.]
 [-13. -30. -20.]
 [  0.  -1.  20.]
 [  5.  78. -19.]]
```

Here you can see that for each of the indices accessed, we updated the corresponding values i.e. the values present for the indices (0,1), (1,2) and (1,1) were updated. In other words, 20.0, 26.0 and 4.5 were replaced with -10, -20 and -30 respectively.

Subset an array using conditions

```
b
```

```
array([[  3., -10.,  99.],
       [-13., -30., -20.],
       [  0.,  -1.,  20.],
       [  5.,  78., -19.]])
```

```
b>0
```

```
array([[ True, False,  True],
       [False, False, False],
       [False, False,  True],
       [ True,  True, False]])
```

On applying a condition on the array 'b', we get an array with boolean values; True where the condition was met, False otherwise. We can use these boolean values, obtained through using conditions, for subsetting the array.

```
b[b>0]
```

```
array([ 3., 99., 20.,  5., 78.])
```

The above statement returns all the elements from the array 'b' which are positive.

Let's say we now want all the positive elements from the third list. Then we need to run the following code:

```
print(b)
b[2]>0
```

```
[[  3. -10.  99.]
 [-13. -30. -20.]
 [  0.  -1.  20.]
 [  5.  78. -19.]]
```

```
array([False, False,  True])
```

When we write b[2]>0, it returns a logical array, returning True wherever the list's value is positive and False otherwise.

Subsetting the list in the following way will, using the condition b[2]>0, will return the actual positive value.

```
b[2,b[2]>0]
```

```
array([20.])
```

Now, what if I want the values from all lists only for those indices where the values in the third list were either 0 or positive.

```
print(b)
print(b[2]>=0)
print(b[:,b[2]>=0])
```

```
[[  3. -10.  99.]
 [-13. -30. -20.]
 [  0.  -1.  20.]
 [  5.  78. -19.]]
[ True False  True]
[[  3.  99.]
 [-13. -20.]
 [  0.  20.]
 [  5. -19.]]
```

For the statement b[:,b[2]>=0], the ':' sign indicates that we are referring to all the lists and the condition 'b[2]>=0' would ensure that we will get the corresponding elements from all the lists which satisfy the condition that the third list is either 0 or positive. In other words, 'b[2]>=0' returns [True, False, True] which will enable us to get the first and the third values from all the lists.

Now lets consider the following scenario, where we want to apply the condition on the third element of each list and then apply the condition across all the elements of the lists:

```
b[:,2]>0
```

```
array([ True, False,  True, False])
```

Here, we are checking whether the third element in each list is positive or not. b[:,2]>0 returns a logical array. Note: it will have as many elements as the number of lists.

```
print(b)
print(b[:,2])
print(b[:,2]>0)
b[b[:,2]>0,:]
```

```
[[  3. -10.  99.]
 [-13. -30. -20.]
 [  0.  -1.  20.]
 [  5.  78. -19.]]
[ 99. -20.  20. -19.]
[ True False  True False]
```

```
array([[  3., -10.,  99.],
       [  0.,  -1.,  20.]])
```

Across the lists, it has extracted those values which correspond to the logical array. Using the statement print(b[:,2]>0), we see that only 99. and 20. are positive, i.e. the third element from each of the first and third lists are positive and hence True. On passing this condition to the array 'b', b[b[:,2]>0,:], we get all those lists wherever the condition evaluated to True i.e. the first and the third lists.

The idea of using numpy is that it allows us to apply functions on multiple values across a full dimension instead of single values. The math package on the other hand works on scalars of single values.

As an example, let's say we wanted to replace the entire 2nd list (index =1) with its absolute values.

We start with importing the math package.

```
import math as m
```

The function exp in the math package returns the exponential value of the number passed as argument.

```
x=-80
m.exp(x)
```

```
1.8048513878454153e-35
```

However, when we pass an array to this function instead of a single scalar value, we get an error.

```
b[1]
```

```
array([-13., -30., -20.])
```

```
b[1]=m.exp(b[1])
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-23-62d051d41c3e> in <module>
----> 1 b[1]=m.exp(b[1])
```

```
TypeError: only size-1 arrays can be converted to Python scalars
```

Basically, the math package converts its inputs to scalars, but since b[1] is an array of multiple elements, it gives an error.

We will need to use the corresponding numpy function to be able to return absolute values of arrays i.e. np.exp().

The following code will return the exponential values of the second list only.

```
print(b)
b[1]=np.exp(b[1])
print(b)
```

```
[[  3. -10.  99.]
 [-13. -30. -20.]
 [  0.  -1.  20.]
 [  5.  78. -19.]]
[[ 3.00000000e+00 -1.00000000e+01  9.90000000e+01]
 [ 2.26032941e-06  9.35762297e-14  2.06115362e-09]
 [ 0.00000000e+00 -1.00000000e+00  2.00000000e+01]
 [ 5.00000000e+00  7.80000000e+01 -1.90000000e+01]]
```

There are multiple such functions available in numpy. We can type 'np.' and press the 'tab' key to see the list of such functions.

All the functions present in the math package will be present in numpy package as well.

Reiterating the advantage of working with numpy instead of math package is that numpy enables us to work with complete arrays. We do not need to write a for loop to apply a function across the array.

axis argument

To understand the axis argument better, we will now explore the 'sum()' function which collapses the array.

```
np.sum(b)
```

```
175.00000226239064
```

Instead of summing the entire array 'b', we want to sum across the list i.e. axis = 0.

```
print(b)
np.sum(b,axis=0)
```

```
[[ 3.00000000e+00 -1.00000000e+01  9.90000000e+01]
 [ 2.26032941e-06  9.35762297e-14  2.06115362e-09]
 [ 0.00000000e+00 -1.00000000e+00  2.00000000e+01]
 [ 5.00000000e+00  7.80000000e+01 -1.90000000e+01]]
```

```
array([  8.00000226,  67.        , 100.        ])
```

If we want to sum all the elements of each list, then we will refer to axis = 1

```
np.sum(b,axis=1)
```

```
array([9.20000000e+01, 2.26239065e-06, 1.90000000e+01, 6.40000000e+01])
```

axis=0 here corresponds to elements across the lists , axis=1 corresponds to within the list elements.

Note: Pandas dataframes, which in a way are 2 dimensional numpy arrays, have each list in a numpy array correspond to a column in pandas dataframe. In a pandas dataframe, axis=0 would refer to rows and axis=1 would refer to columns.

Now we will go through some commonly used numpy functions. We will use the rarely used functions as and when we come across them.

The commonly used functions help in creating special kinds of numpy arrays.

arange()

```
np.arange(0,6)
```

```
array([0, 1, 2, 3, 4, 5])
```

The arange() function returns an array starting from 0 until (6-1) i.e. 5.

```
np.arange(2,8)
```

```
array([2, 3, 4, 5, 6, 7])
```

We can also control the starting and ending of an arange array. The above arange function starts from 2 and ends with (8-1) i.e. 7, incrementing by 1.

The arange function is used for creating a sequence of integers with different starting and ending points having an increment of 1.

linspace()

To create a more customized sequence we can use the linspace() function. The argument num gives the number of elements in sequence. The elements in the sequence will be equally spaced.

```
np.linspace(start=2,stop=10,num=15)
```

```
array([ 2.        ,  2.57142857,  3.14285714,  3.71428571,  4.28571429,
        4.85714286,  5.42857143,  6.        ,  6.57142857,  7.14285714,
        7.71428571,  8.28571429,  8.85714286,  9.42857143, 10.        ])
```

random.randint()

Given an array of numbers, we can randomly sample elements from that array using the randint() function from the random package.

```
np.random.randint(high=10,low=1,size=(2,3))
```

```
array([[4, 5, 8],
       [1, 3, 4]])
```

The above code creates a random array of size (2,3) i.e. two lists having three elements each. These random elements are chosen from numbers between 1 and 10.

random.random()

We can also create an array of random numbers using the random() function from the random package.

```
np.random.random(size=(3,4))
```

```
array([[0.83201385, 0.50934665, 0.04015334, 0.71826046],
       [0.25727761, 0.26890597, 0.71662683, 0.23890495],
       [0.17166143, 0.08512553, 0.43757489, 0.89178915]])
```

The above random() function creates an array of size (3,4) where the elements are real numbers between 0 to 1.

random.choice()

Consider an array ranging from 0 to 9:

```
x = np.arange(0, 10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.random.choice(x,6)
```

```
array([9, 0, 4, 3, 8, 5])
```

random.choice() functions helps us to select 6 random numbers from the input array x. Every time we run this code, the result will be different.

We can see that, at times, the function ends up picking up the same element twice. If we want to avoid that i.e. get each number only once or in other words, get the elements without replacement; then we need to set the argument 'replace' as False which is True by default.

```
np.random.choice(x,6,replace=False)
```

```
array([4, 0, 7, 1, 8, 6])
```

Now when we run the above code again, we will get different values, but we will not see any number more than once.

We will also use the random.choice() function to simulate data often.

```
y = np.random.choice(['a','b'], 6)
print(y)
```

```
['b' 'a' 'b' 'b' 'a' 'b']
```

The code above picks 'a' or 'b' randomly 6 times.

```
y = np.random.choice(['a','b'], 1000)
```

The code above samples 'a' and 'b' a 1000 times. Now if we take unique values from this array, it will be 'a' and 'b', as shown by the code below. The return_counts argument gives the number of times the two elements are present in the array created.

```
np.unique(y, return_counts=True)
```

```
(array(['a', 'b'], dtype='<U1'), array([509, 491], dtype=int64))
```

By default, both 'a' and 'b' get picked up with equal probability in the random sample. This does not mean that the individual values are not random; but the overall percentage of 'a' and 'b' remains almost the same.

However, if we want the two values according to a specific proportion, we can use the argument 'p' in the random.choice() function.

```
y=np.random.choice(['a','b'],1000,p=[0.8,0.2])
```

```
np.unique(y, return_counts=True)
```

```
(array(['a', 'b'], dtype='<U1'), array([797, 203], dtype=int64))
```

Now we can observe that the value 'a' is present aprroximately 80% of the time and value 'b' appears around 20% of the time. the individual values are still random, though overall, 'a' will appear 80% of the time as specified and 'b' will appear 20% of the time. Since the underlying process is inherently random, these probabilities will be close to 80% and 20% respectively but may not be exactly the same. In fact, if we draw samples of smaller sizes, the difference could be quite wide as shown in the code below.

```
y=np.random.choice(['a','b'],10,p=[0.8,0.2])
np.unique(y, return_counts=True)
```

```
(array(['a', 'b'], dtype='<U1'), array([9, 1], dtype=int64))
```

Here we sample only 10 values in the proportion 8:2. As you repeat this sampling process, at times the proportion may match, but many times the difference will be big. As we increase the size of the sample, the number of samples drawn for each element will be closer to the proportion specified.

sort()

To understand this function, lets create a single dimensional array:

```
x=np.random.randint(high=100,low=12,size=(15,))
print(x)
```

```
[89 60 31 82 97 92 14 96 75 12 36 37 25 27 62]
```

The array contains 15 random integers ranging from 12 to 100.

```
x.sort()
```

Sorting happens in place and it does not return anything.

```
print(x)
```

```
[12 14 25 27 31 36 37 60 62 75 82 89 92 96 97]
```

x.sort() sorts the entire array in an ascending order.

Now let us see how sorting works with 2 dimensional arrays.

```
x=np.random.randint(high=100,low=12,size=(4,6))
```

```
x
```

```
array([[61, 60, 78, 20, 56, 50],
       [27, 56, 88, 69, 40, 26],
       [35, 83, 40, 17, 74, 67],
       [33, 78, 25, 19, 53, 12]])
```

This array is 2 dimensional containing 4 lists and each list has 6 elements.

If we use the sort function directly, the correspondence between the elements is broken i.e. each individual list is sorted independent of the other lists. We may not want this. The first elements in each list belong together, so do the second and so on; but after sorting this correspondence is broken.

```
np.sort(x)
```

```
array([[20, 50, 56, 60, 61, 78],
       [26, 27, 40, 56, 69, 88],
       [17, 35, 40, 67, 74, 83],
       [12, 19, 25, 33, 53, 78]])
```

argsort()

In order to take care of this, we may use argsort() as follows:

For maintaining order along either of the axis, we can extract indices of the sorted values and reorder original array with these indices. Lets see the code below:

```
print(x)
x[:,2]
```

```
[[61 60 78 20 56 50]
 [27 56 88 69 40 26]
 [35 83 40 17 74 67]
 [33 78 25 19 53 12]]
```

```
array([78, 88, 40, 25])
```

These return the third element from each list of the 2 dimensional array x.

Lets say we want to sort the 2 dimensional array x by these values and the other values should move with them maintaining the correspondence. This is where we will use argsort() function.

```
print(x[:,2])
x[:,2].argsort()
```

```
[78 88 40 25]
```

```
array([3, 2, 0, 1], dtype=int64)
```

Instead of sorting the array, argsort() returns the indices of the elements after they are sorted.

The value with index 3 i.e. 25 should appear first, the value with index 2 i.e. 40 should appear next and so on.

We can now pass these indices to arrange all the lists according to them. We will observe that the correspondence does not break.

```
x[x[:,2].argsort(),:]
```

```
array([[33, 78, 25, 19, 53, 12],
       [35, 83, 40, 17, 74, 67],
       [61, 60, 78, 20, 56, 50],
       [27, 56, 88, 69, 40, 26]])
```

All the lists have been arranged according to order given by x[:,2].argsort() for the third element across the lists.

max() and argmax() - we can similarly use min() and argmin()

```
x=np.random.randint(high=100,low=12,size=(15,))
```

```
x
```

```
array([17, 77, 49, 36, 39, 27, 63, 99, 94, 22, 55, 66, 93, 32, 16])
```

```
x.max()
```

```
99
```

The max() function will simply return the maximum value from the array.

```
x.argmax()
```

```
7
```

The argmax() function on the other hand will simply give the index of the maximum value i.e. the maximum number 99 lies at the index 7.

Pandas

Introduction to Pandas

In this module we will start with the python package named pandas which is primarily used for handling datasets in python. We will cover the following topics:

1. Manually creating data frames
2. Reading data from external file
3. Peripheral summary of the data
4. Subsetting on the basis of row number and column number

5. Subsetting on the basis of conditions
6. Subsetting on the basis of column names

We start with importing the pandas package

```
import pandas as pd
import numpy as np
import random
```

1. Manually creating dataframes

There are two ways of creating dataframes:

1. From lists
2. From dictionary

1. Creating dataframe using lists:

We will start with creating some lists and then making a dataframe using these lists.

```
age=np.random.randint(low=16,high=80,size=[20,])
city=np.random.choice(['Mumbai','Delhi','Chennai','Kolkata'],20)
default=np.random.choice([0,1],20)
```

We can zip these lists to convert them to a single list tuples. Each tuple in the list will refer to a row in the dataframe.

```
mydata=list(zip(age,city,default))
```

```
mydata
```

```
[(33, 'Mumbai', 0),
 (71, 'Kolkata', 1),
 (28, 'Mumbai', 1),
 (46, 'Mumbai', 1),
 (22, 'Delhi', 1),
 (23, 'Delhi', 0),
 (58, 'Mumbai', 0),
 (60, 'Kolkata', 1),
 (69, 'Mumbai', 1),
 (69, 'Kolkata', 1),
 (34, 'Mumbai', 0),
 (70, 'Delhi', 0),
 (44, 'Chennai', 1),
 (30, 'Delhi', 1),
 (62, 'Delhi', 0),
 (19, 'Mumbai', 1),
 (68, 'Mumbai', 1),
 (74, 'Chennai', 1),
 (41, 'Mumbai', 1),
 (24, 'Chennai', 1)]
```

Each of the tuples come from zipping the elements in each of the lists (age, city and default) that we created earlier.

Note: You may have different values when you run this code since we are randonly generating the lists using the random package.

We can then put this list of tuples in a dataframe simply by using the pd.DataFrame function.

```
df = pd.DataFrame(mydata, columns=["age", "city", "default"])
```

```
df.head() # we are using head() function which displays only the first 5 rows.
```

	age	city	default
0	33	Mumbai	0
1	71	Kolkata	1
2	28	Mumbai	1
3	46	Mumbai	1
4	22	Delhi	1

As you can observe, this is a simple dataframe with 3 columns and 20 rows, having the three lists: age, city and default as columns. The column names could have been different too, they do not have to necessarily match the list names.

2. Creating dataframe from a dictionary

Another way of creating dataframes is using a dictionary. Here we will need to provide the column names separately, which will be picked as the values of the key.

```
df = pd.DataFrame({"age":age, "city":city, "default":default})
```

Here the key values ("age", "city" and "default") will be taken as column names and the lists (age, city and default) will contain the values themselves.

```
df.head() # we are using head() function which displays only the first 5 rows.
```

	age	city	default
0	33	Mumbai	0
1	71	Kolkata	1
2	28	Mumbai	1
3	46	Mumbai	1
4	22	Delhi	1

In both the cases i.e. creating the dataframe using list or dictionary, the resultant dataframe is the same. The process of creating them is different but there is no difference in the resulting dataframes.

2. Reading data from external file

We can read data from external files using the pandas function read_csv().

Lets first create a string containing the path to the .csv file.

```
file=r'/Users/anjal/Dropbox/0.0 Data/loans data.csv'
```

Here, 'r' is added at the beginning of the path. This is to ensure that the file path is read as a raw string and any special character combinations are not interpreted as their special meaning by Python. e.g. \n means newline, which will be ignored from the file path when we add 'r' at the beginning of the string.

```
ld = pd.read_csv(file)
```

The pandas function read_csv() reads the file present in the path given by the agrument 'file'.

3. Peripheral summary of the data

```
ld.head()
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpose	Debt.To.Income.Ratio
0	81174.0	20000	20000	8.90%	36 months	debt_consolidation	14.90%
1	99592.0	19200	19200	12.12%	36 months	debt_consolidation	28.36%
2	80059.0	35000	35000	21.98%	60 months	debt_consolidation	23.81%
3	15825.0	10000	9975	9.99%	36 months	debt_consolidation	14.30%
4	33182.0	12000	12000	11.71%	36 months	credit_card	18.78%

The head() function of the pandas dataframe created, by default, returns the top 5 rows of the dataframe. If we wish to see more or less rows, for instance 10 rows, then we will pass the number as an argument to the head() function.

```
ld.head(10)
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpose	Debt.To.Income.Ratio
0	81174.0	20000	20000	8.90%	36 months	debt_consolidation	14.90%
1	99592.0	19200	19200	12.12%	36 months	debt_consolidation	28.36%
2	80059.0	35000	35000	21.98%	60 months	debt_consolidation	23.81%
3	15825.0	10000	9975	9.99%	36 months	debt_consolidation	14.30%
4	33182.0	12000	12000	11.71%	36 months	credit_card	18.78%
5	62403.0	6000	6000	15.31%	36 months	other	20.05%
6	48808.0	10000	10000	7.90%	36 months	debt_consolidation	26.09%
7	22090.0	33500	33450	17.14%	60 months	credit_card	14.70%
8	76404.0	14675	14675	14.33%	36 months	credit_card	26.92%
9	15867.0	.	7000	6.91%	36 months	credit_card	7.10%

Here, we get the top 10 rows of the dataframe.

We can get the column names by using the 'columns' attribute of the pandas dataframe.

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
       'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'State',
       'Home.Ownership', 'Monthly.Income', 'FICO.Range', 'Open.CREDIT.Lines',
       'Revolving.CREDIT.Balance', 'Inquiries.in.the.Last.6.Months',
       'Employment.Length'],
      dtype='object')
```

If we want to see the type of these columns, we can use the attribute 'dtypes' of the pandas dataframe.

```
ld.dtypes
```

```
ID                              float64
Amount.Requested                 object
Amount.Funded.By.Investors       object
Interest.Rate                    object
Loan.Length                      object
Loan.Purpose                     object
Debt.To.Income.Ratio             object
State                            object
Home.Ownership                   object
Monthly.Income                  float64
FICO.Range                       object
Open.CREDIT.Lines                object
Revolving.CREDIT.Balance         object
Inquiries.in.the.Last.6.Months  float64
Employment.Length                object
dtype: object
```

The float64 datatype refers to numeric columns and object datatype refers to categorical columns.

If we want a concise summary of the dataframe including information about null values, we use the info() function of the pandas dataframe.

```
ld.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2500 entries, 0 to 2499
Data columns (total 15 columns):
ID                              2499 non-null float64
Amount.Requested                2499 non-null object
Amount.Funded.By.Investors      2499 non-null object
Interest.Rate                   2500 non-null object
Loan.Length                     2499 non-null object
Loan.Purpose                    2499 non-null object
Debt.To.Income.Ratio            2499 non-null object
State                           2499 non-null object
Home.Ownership                  2499 non-null object
Monthly.Income                  2497 non-null float64
FICO.Range                      2500 non-null object
Open.CREDIT.Lines               2496 non-null object
```

```
Revolving.CREDIT.Balance          2497 non-null object
Inquiries.in.the.Last.6.Months    2497 non-null float64
Employment.Length                 2422 non-null object
dtypes: float64(3), object(12)
memory usage: 293.0+ KB
```

If we want to get the dimensions i.e. numbers of rows and columns of the data, we can use the attribute 'shape'.

```
ld.shape
```

```
(2500, 15)
```

This dataframe has 2500 rows or observations and 15 columns.

4. Subsetting on the basis of row number and column number

We will now start with subsetting data on the basis of row and column numbers. The count starts with 0 for both rows and columns.

```
ld1=ld.iloc[3:7,1:5]
ld1
```

	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length
3	10000	9975	9.99%	36 months
4	12000	12000	11.71%	36 months
5	6000	6000	15.31%	36 months
6	10000	10000	7.90%	36 months

'iloc' refers to subsetting the dataframe by position. Here we have extracted the rows from 3rd to the 6th (7-1) position and columns from 1st to 4th (5-1) position.

To understand this further, we will further subset the 'ld1' dataframe. It currently has 4 rows and 4 columns. The indexes (3, 4, 5 and 6) come from the original dataframe. Lets subset the 'ld1' dataframe further.

```
ld1.iloc[2:4,1:3]
```

	Amount.Funded.By.Investors	Interest.Rate
5	6000	15.31%
6	10000	7.90%

You can see here that the positions are relative to the current dataframe 'ld1' and not the original dataframe 'ld'. Hence we end up with the 3rd and 4th rows along with 2nd and 3rd columns of the new dataframe 'ld1' and not the original dataframe 'ld'.

Generally, we do not subset dataframes by position. We normally subset the dataframes using conditions or column names.

5. & 6. Subsetting on the basis of conditions and columns

When we subset on the basis of conditions or column names, we can directly pass the conditions or column names in square brackets.

Lets say, we want to subset the dataframe and get only those rows for which the 'Home.Ownership' is of the type 'MORTGAGE' and the 'Monthly.Income' is above 5000.

Note: When we combine multiple conditions, we have to enclose them in parenthesis else your results will not be as expected.

```
ld[(ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)].head()
# we are using head() function which displays only the first 5 rows.
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpose	Debt.To.Income.Rati
0	81174.0	20000	20000	8.90%	36 months	debt_consolidation	14.90%
2	80059.0	35000	35000	21.98%	60 months	debt_consolidation	23.81%
7	22090.0	33500	33450	17.14%	60 months	credit_card	14.70%
12	41200.0	28000	27975	21.67%	60 months	debt_consolidation	13.07%
20	86099.0	22000	21975	21.98%	36 months	debt_consolidation	11.19%

On observing the results, you will notice that for each of the rows both the conditions will be satisfied i.e. 'Home.Ownership' will be 'MORTGAGE' and the 'Monthly.Income' will be greater than 5000.

In case we want to access a single columns data only, then we simply have to pass the column name in square brackets as follows:

```
ld['Home.Ownership'].head()
```

```
0    MORTGAGE
1    MORTGAGE
2    MORTGAGE
3    MORTGAGE
4        RENT
Name: Home.Ownership, dtype: object
```

However, if we want to access multiple columns, then the names need to be passed as a list. For instance, if we wanted to extract both 'Home.Ownership' and 'Monthly.Income', we would need to pass it as a list, as follows:

```
ld[['Home.Ownership','Monthly.Income']].head()
# note the double sqaure brackets used to subset the dataframe using multiple columns
```

	Home.Ownership	Monthly.Income
0	MORTGAGE	6541.67
1	MORTGAGE	4583.33
2	MORTGAGE	11500.00
3	MORTGAGE	3833.33
4	RENT	3195.00

If we intend to use both, condition as well as column names, we will need to use the .loc with the pandas dataframe name.

Observing the code below, we subset the dataframe using conditions and columns both. We are subsetting the rows, using the condtition '(ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)' and we extract the 'Home.Ownership' and 'Monthly.Income' columns.

Here, both the conditions should be met; to get that observation in the output. e.g. We can see in the first row of the dataframe that 'Home.Ownership' is 'MORTGAGE', the 'Monthly.Income' is more than 5000. If either of the condition is false, we will not see that observation in the resulting dataframe.

```
ld.loc[(ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000),
['Home.Ownership','Monthly.Income']].head()
```

	Home.Ownership	Monthly.Income
0	MORTGAGE	6541.67
2	MORTGAGE	11500.00
7	MORTGAGE	13863.42
12	MORTGAGE	14166.67
20	MORTGAGE	6666.67

The resulting dataframe has only 2 columns and 686 rows. The rows correspond to the result obtained when the condition (ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000) is applied to the 'ld' dataframe.

Here, we have extracted rows on the basis of some conditions.

What if we wanted to subset only those rows which did not satisfy a condition i.e. we want to negate a condition. In order to do this, we can put a '~' sign before the condition.

In the following code, we will get all the observations that do not satisfy the condition ((ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)). Here, both the conditions should not be met; if either of them holds true, then we will get that observation in the output. e.g. Considering the first row, even though the 'Monthly.Income' is less than 5000 (i.e. one of the conditions is not met), since the other condition of 'Home.Ownership' being equal to 'MORTGAGE' holds true - the entire condition ((ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)) is not negated and hence we see this observation in the output.

```
ld.loc[~((ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)),
['Home.Ownership','Monthly.Income']].head()
```

	Home.Ownership	Monthly.Income
1	MORTGAGE	4583.33
3	MORTGAGE	3833.33
4	RENT	3195.00
5	OWN	4891.67
6	RENT	2916.67

In short, the '~' sign gives us rest of the observations by negating the condition.

Drop columns on the basis of names

To drop columns on the basis of column names, we can use the in-built drop() function.

In order to drop the columns, we pass the list of columns to be dropped along with specifying the axis argument as 1 (since we are dropping columns).

The following code will return all the columns except 'Home.Ownership' and 'Monthly.Income'.

```
ld.drop(['Home.Ownership','Monthly.Income'],axis=1).head()
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpose	Debt.To.Income.Ratio
0	81174.0	20000	20000	8.90%	36 months	debt_consolidation	14.90%
1	99592.0	19200	19200	12.12%	36 months	debt_consolidation	28.36%
2	80059.0	35000	35000	21.98%	60 months	debt_consolidation	23.81%
3	15825.0	10000	9975	9.99%	36 months	debt_consolidation	14.30%
4	33182.0	12000	12000	11.71%	36 months	credit_card	18.78%

However, when we check the columns of the 'ld' dataframe now, the two columns which we presumably deleted, are still there.

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
       'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'State',
       'Home.Ownership', 'Monthly.Income', 'FICO.Range', 'Open.CREDIT.Lines',
       'Revolving.CREDIT.Balance', 'Inquiries.in.the.Last.6.Months',
       'Employment.Length'],
      dtype='object')
```

Basically, ld.drop(['Home.Ownership','Monthly.Income'],axis=1) did not change the original dataframe and the deleted columns are still present in 'ld' dataframe.

What happens is that the ld.drop() function gives us an output; it does not make any inplace changes.

So, in case we wish to delete the columns from the original dataframe, we can do two things:

- Equate the output to the original dataframe
- Set the 'inplace' argument of the drop() function to True

We can update the original dataframe by equating the output to the original dataframe as follows:

```
ld=ld.drop(['Home.Ownership','Monthly.Income'],axis=1)
```

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
       'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'State',
       'FICO.Range', 'Open.CREDIT.Lines', 'Revolving.CREDIT.Balance',
       'Inquiries.in.the.Last.6.Months', 'Employment.Length'],
      dtype='object')
```

The second way to update the original dataframe is to set the 'inplace' argument of the drop() function to True

```
ld.drop(['State'],axis=1,inplace=True)
```

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
       'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'FICO.Range',
       'Open.CREDIT.Lines', 'Revolving.CREDIT.Balance',
       'Inquiries.in.the.Last.6.Months', 'Employment.Length'],
      dtype='object')
```

Now you will notice that the deleted columns are not present in the original dataframe 'ld' anymore.

We need to be careful when using the inplace=True option; the function drop() doesnt output anything. So we should not equate ld.drop(['State'],axis=1,inplace=True) to the original dataframe. If we equate it to the original dataframe 'ld', then 'ld' will end up as a None type object.

Drop columns using del keyword

We can also delete a column using the 'del' keyword. The following code will remove the column 'Employment.Length' from the original dataframe 'ld'.

```
del ld['Employment.Length']
```

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
       'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'FICO.Range',
       'Open.CREDIT.Lines', 'Revolving.CREDIT.Balance',
       'Inquiries.in.the.Last.6.Months'],
      dtype='object')
```

On checking the columns of the 'ld' dataframe, we can observe that 'Employment.Length' column is not present.

Modifying data with Pandas

Now we will be covering how to modify dataframes using pandas. Note: We will be using a simulated dataset to understand this, as all data pre processing opportunities might not be present in a single real world dataset. However, the various techniques that we'll learn here will be useful in different datasets that we come across. Some of the techniques that we cover are as follows:

1. Changing variable types
2. Adding/modifying variables with algebraic operations
3. Adding/modifying variables based on conditions
4. Handling missing values
5. Creating flag variables
6. Creating multiple columns from a variable separated by a delimiter

Lets start with importing the packages we need:

```
import numpy as np
import pandas as pd
```

We will start with creating a custom dataframe having 7 columns and 50 rows as follows:

```
age=np.random.choice([15,20,30,45,12,'10',15,'34',7,'missing'],50)
fico=np.random.choice(['100-150','150-200','200-250','250-300'],50)
city=np.random.choice(['Mumbai','Delhi','Chennai','Kolkata'],50)
ID=np.arange(50)
rating=np.random.choice(['Excellent','Good','Bad','Pathetic'],50)
balance=np.random.choice([10000,20000,30000,40000,np.nan,50000,60000],50)
children=np.random.randint(high=5,low=0,size=(50,))
```

```
mydata=pd.DataFrame({'ID':ID,'age':age,'fico':fico,'city':city,'rating':rating,'balance':balance,'children'
:children})
```

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children
0	0	12	250-300	Chennai	Excellent	10000.0	3
1	1	15	150-200	Chennai	Bad	20000.0	3
2	2	missing	250-300	Chennai	Pathetic	20000.0	2
3	3	45	250-300	Delhi	Bad	NaN	3
4	4	missing	250-300	Delhi	Pathetic	50000.0	2

1. Changing variable types

Lets check the datatypes of the columns in the mydata dataframe.

```
mydata.dtypes
```

```
ID            int32
age          object
fico         object
city         object
rating       object
balance     float64
children      int32
dtype: object
```

We can see that 'age' column is of the object datatype, though it should have been numeric, maybe due to some character values in the column. We can change the datatype to 'numeric'; the character values which cannot be changed to numeric will be assigned missing values i.e. NaN's automatically.

There are multiple numeric formats in Python e.g. integer, float, unsigned integer etc. The to_numeric() functions chooses the best one for the column under consideration.

```
mydata['age']=pd.to_numeric(mydata['age'])
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

pandas/_libs/src\inference.pyx in pandas._libs.lib.maybe_convert_numeric()
```

```
ValueError: Unable to parse string "missing"
```

```
During handling of the above exception, another exception occurred:
```

```
ValueError                                Traceback (most recent call last)

<ipython-input-95-b9b69f73c0ff> in <module>
----> 1 mydata['age']=pd.to_numeric(mydata['age'])
```

```
~\Anaconda3\lib\site-packages\pandas\core\tools\numeric.py in to_numeric(arg, errors, downcast)
    131             coerce_numeric = False if errors in ('ignore', 'raise') else True
    132             values = lib.maybe_convert_numeric(values, set(),
--> 133                                                coerce_numeric=coerce_numeric)
    134
    135     except Exception:
```

```
pandas/_libs/src\inference.pyx in pandas._libs.lib.maybe_convert_numeric()
```

```
ValueError: Unable to parse string "missing" at position 2
```

When we run the code above, we get an error i.e. "Unable to parse string "missing" at position 2". This error means that there are a few values in the column that cannot be converted to numbers; in our case its the value 'missing' which cannot be converted to a number. In order to handle this, we need to set the errors argument of the to_numeric() function to 'coerce' i.e. errors='coerce'. When we use this argument, wherever it was not possible to convert the values to numeric, it converted them to missing values i.e. NaN's.

```
mydata['age']=pd.to_numeric(mydata['age'], errors='coerce')
```

```
mydata['age'].head()
```

```
0    12.0
1    15.0
2     NaN
3    45.0
4     NaN
Name: age, dtype: float64
```

As we can observe in the rows 2,4,etc, wherever there was the 'missing' string present, which could not be converted to numbers are now converted to NaN's or missing values.

2. Adding/modifying variables with algebraic operations

Now lets look at some additions and modifications of columns using algebraic operations.

Adding a column with constant value

```
mydata['const_var']=100
```

The above code adds a new column 'const_var' to the mydata dataframe and each element in that column is 100.

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var
0	0	12.0	250-300	Chennai	Excellent	10000.0	3	100
1	1	15.0	150-200	Chennai	Bad	20000.0	3	100
2	2	NaN	250-300	Chennai	Pathetic	20000.0	2	100
3	3	45.0	250-300	Delhi	Bad	NaN	3	100
4	4	NaN	250-300	Delhi	Pathetic	50000.0	2	100

Apply function on entire columns

If we want to apply a function on an entire column of a dataframe, we use a numpy function; e.g log as shown below:

```
mydata['balance_log']=np.log(mydata['balance'])
```

The code above creates a new column 'balance_log' which has the logarithmic value of each element present in the 'balance' column. A numpy function np.log() is used to do this.

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log
0	0	12.0	250-300	Chennai	Excellent	10000.0	3	100	9.210340
1	1	15.0	150-200	Chennai	Bad	20000.0	3	100	9.903488
2	2	NaN	250-300	Chennai	Pathetic	20000.0	2	100	9.903488
3	3	45.0	250-300	Delhi	Bad	NaN	3	100	NaN
4	4	NaN	250-300	Delhi	Pathetic	50000.0	2	100	10.819778

Add columns using complex algebraic calculations

We can do many complex algebraic calculations as well to create/add new columns to the data.

```
mydata['age_children_ratio']=mydata['age']/mydata['children']
```

The code above creates a new column 'age_children_ratio'; each element of which will be the result of the division of the corresponding elements present in the 'age' and 'children' columns.

```
mydata.head(10)
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_children_ratio
0	0	12.0	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.0	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	NaN	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	NaN
3	3	45.0	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	NaN	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	NaN
5	5	20.0	100-150	Delhi	Pathetic	60000.0	3	100	11.002100	6.666667
6	6	34.0	100-150	Mumbai	Good	40000.0	0	100	10.596635	inf
7	7	20.0	200-250	Kolkata	Pathetic	30000.0	4	100	10.308953	5.000000
8	8	20.0	150-200	Mumbai	Good	20000.0	3	100	9.903488	6.666667
9	9	20.0	150-200	Chennai	Bad	50000.0	4	100	10.819778	5.000000

Notice that when a missing value is involved in any calculation, the result is also a missing value. We observe that in the 'age_children_ratio' column we have both NaN's (missing values) as well as inf (infinity). We get missing values in the 'age_children_ratio' column wherever 'age' has missing values and we get 'inf' wherever the number of children is 0 and we end up dividing by 0.

Handling missing values

Lets say we did not want missing values involved in the calculation i.e. we want to impute the missing values before computing the 'age_children_ratio' column. For this we would first need to identify the missing values. The isnull() function will give us a logical array which can be used to isolate missing values and update these with whatever value we want to impute with.

```
mydata['age'].isnull().head()
```

```
0    False
1    False
2     True
3    False
4     True
Name: age, dtype: bool
```

In the outcome of the code above, we observe that wherever there is a missing value the corresponding logical value is True.

If we want to know the number of missing values, we can sum the logical array as follows:

```
mydata['age'].isnull().sum()
```

```
5
```

We have 5 missing values in the 'age' column.

The following code returns only those elements where the 'age' column has missing values.

```
mydata.loc[mydata['age'].isnull(),'age']
```

```
2    NaN
4    NaN
17   NaN
36   NaN
38   NaN
Name: age, dtype: float64
```

One of the ways of imputing the missing values is with mean. Once these values are imputed, we then carry out the calculation done above.

```
mydata.loc[mydata['age'].isnull(),'age'] = np.mean(mydata['age'])
```

In the code above, using the loc() function of the dataframe, on the row side we first access those rows where the 'age' column is null and on the column side we access only the 'age' column. In other words, all the missing values from the age column will be replaced with the mean computed using the 'age' column.

```
mydata['age'].head()
```

```
0    12.000000
1    15.000000
2    19.533333
3    45.000000
4    19.533333
Name: age, dtype: float64
```

The missing values from the 'age' column has been replaced by the mean of the column i.e. 19.533333.

Now, we can compute the 'age_children_ratio' again; this time without missing values. We will observe that there are no missing values in the newly created column now. We however, observe inf's i.e. infinity which occurs wherever we divide by 0.

```
mydata['age_children_ratio']=mydata['age']/mydata['children']
```

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_children_ratio
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667

3. Adding/modifying variables based on conditions

Now we will see how do we add or modify variables based on conditions.

Lets say we want to replace the 'rating' column values with some numeric score - {'pathetic' : -1 , 'bad' : 0 , 'good or excellent': 1}. We can do it using the np.replace() function as follows:

```
mydata['rating_score']=np.where(mydata['rating'].isin(['Good','Excellent']),1,0)
```

Using the above code, we create a new column 'rating_score' and wherever either 'Good' or 'Excellent' is present, we replace it with a 1 else with a 0 as we can see below. The function isin() is used when we need to consider mulitple values; in our case 'Good' and 'Excellent'.

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_children_ratio	rating_score
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000	1
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000	0
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667	0
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000	0
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667	0

An alternative way of updating the 'rating' column will be as follows:

```
mydata.loc[mydata['rating']=='Pathetic','rating_score']=-1
```

In the code above, wherever the value in 'rating' column is 'Pathetic', we update the 'rating_score' column to -1 and leave the rest as is. The above code could be been written using np.where() function as well. The np.where() function is similar to the ifelse statement we may have seen in other languages.

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_children_ratio	rating_score
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000	1
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000	0
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667	-1
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000	0
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667	-1

Now we can see that the 'rating_score' column takes the values 0, 1 and -1 depending on the corresponding values from the 'rating' column.

At times, we may have columns which we may want to split into multiple columns; column 'fico' in our case. One sample value is '100-150'. The datatype for 'fico' is considered as object. Its a difficult problem to solve if we use a for loop for processing each value. However, we will discuss an easier approach which will be very useful to know when pre-processing your data.

Coming back to the 'fico' column, one of the first thing that comes to mind when we want to separate the values from 'fico' column into multiple columns is the split() function. The split function works on strings, but the current datatype of 'fico' column is object; object type does not understand string functions. If we apply the split() function directly on 'fico', we will get the following error.

```
mydata['fico'].split()
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-117-e967df8ee134> in <module>
----> 1 mydata['fico'].split()
```

```
~\Anaconda3\lib\site-packages\pandas\core\generic.py in __getattr__(self, name)
   4374             if self._info_axis._can_hold_identifiers_and_holds_name(name):
   4375                 return self[name]
-> 4376             return object.__getattribute__(self, name)
   4377
   4378     def __setattr__(self, name, value):
```

```
AttributeError: 'Series' object has no attribute 'split'
```

In order to handle this, we will first need to extract the 'str' attribute of the 'fico' column and then apply the split() function. This will be the case for all string functions and not just split().

```
mydata['fico'].str.split("-").head()
```

```
0    [250, 300]
1    [150, 200]
2    [250, 300]
3    [250, 300]
4    [250, 300]
Name: fico, dtype: object
```

We can see that each of the elements have been split on the basis of '-'. However, it is still present in a single column. We need the values in separate columns. We will set the 'expand' argument of the split() function to True in order to handle this.

```
mydata['fico'].str.split("-",expand=True).head()
```

	0	1
0	250	300
1	150	200
2	250	300
3	250	300
4	250	300

This code by default creates a dataframe.

```
k=mydata['fico'].str.split("-",expand=True).astype(float)
```

Notice that we have converted the columns to float using the astype(float) function; since after splitting, by default, the datatype of each column created would be object. But we want to consider each column as numeric datatype, hence the columns are converted to float. Converting to float is not a required step when splitting columns. We do it only because these values are supposed to be considered numeric in the current context.

We can now access the individual columns using k[0] or k[1].

```
k[0].head()
```

```
0    250.0
1    150.0
2    250.0
3    250.0
4    250.0
Name: 0, dtype: float64
```

We can either concatenate this dataframe to the 'mydata' dataframe after giving proper header to both the columns or we can directly assign two new columns in the 'mydata' dataframe as follows:

```
mydata['f1'],mydata['f2']=k[0],k[1]
```

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_children_ratio	rating_score
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000	1
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000	0
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667	-1
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000	0
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667	-1

We do not need the 'fico' column anymore as we have its values in tow separate columns; hence we will delete it.

```
del mydata['fico']
```

```
mydata.head()
```

	ID	age	city	rating	balance	children	const_var	balance_log	age_children_ratio	rating_score	f1
0	0	12.000000	Chennai	Excellent	10000.0	3	100	9.210340	4.000000	1	250.0
1	1	15.000000	Chennai	Bad	20000.0	3	100	9.903488	5.000000	0	150.0
2	2	19.533333	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667	-1	250.0
3	3	45.000000	Delhi	Bad	NaN	3	100	NaN	15.000000	0	250.0
4	4	19.533333	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667	-1	250.0

Converting categorical variables to flag variables or dummies

For all the machine algorithm that we will encounter, we will need to convert categorical variables to flag variables. If a variable has n distinct categories, we will need to create (n-1) flag variables. We cannot use categorical valriables - character type variables - object type variables as is in the data that we pass to the machine learning algorithms.

Manually, dummies can be created as follows:

Lets consider the city variable.

```
print(mydata['city'].unique())
print(mydata['city'].nunique())
```

```
['Chennai' 'Delhi' 'Mumbai' 'Kolkata']
4
```

It consists of 4 unique elements - 'Kolkata', 'Mumbai', 'Chennai', 'Delhi'. For this variable, we will need to create three dummies.

The code below creates a flag variable when the 'city' column has the value 'Mumbai'.

```
mydata['city_mumbai']=np.where(mydata['city']=='Mumbai',1,0)
```

Wherever the variable 'city' takes the value 'Mumbai', the flag valriable 'city_mumbai' will be 1 otherwise 0.

There is another way to do this, where we write the condition and convert the logical value to integer.

```
mydata['city_chennai']=(mydata['city']=='Chennai').astype(int)
```

Code "mydata['city']=='Chennai'" gives a logical array; wherever the city is 'Chennai', the value on 'city_chennai' flag variable is True, else False.

```
(mydata['city']=='Chennai').head()
```

```
0     True
1     True
2     True
3    False
4    False
Name: city, dtype: bool
```

When we convert it to an integer, wherever there was True, we get a 1 and wherever there was False, we get a 0.

```
((mydata['city']=='Chennai').astype(int)).head()
```

```
0    1
1    1
2    1
3    0
4    0
Name: city, dtype: int32
```

We can use either of the methods for creating flag variables, the end result is same.

We need to create one more flag variable.

```
mydata['city_kolkata']=np.where(mydata['city']=='Kolkata',1,0)
```

Once the flag variables have been created, we do not need the original variable i.e. we do not need the 'city' variable anymore.

```
del mydata['city']
```

This way of creating dummies requires a lot of coding, even if we somehow use a for loop. As an alternative, we can use get_dummies() function from pandas directly to do this.

We will create dummies for the variable 'rating' using this method.

```
print(mydata['rating'].unique())
print(mydata['rating'].nunique())
```

```
['Excellent' 'Bad' 'Pathetic' 'Good']
4
```

This too has 4 unique values.

The pandas function which creates dummies is get_dummies() in which we pass the column for which dummies need to be created. By default, the get_dummies function creates n dummies if n unique values are present in the column i.e. for 'rating' column, by default, get_dummies function, creates 4 dummy variables. We do not want that. Hence, we pass the argument 'drop_first=True' which removes one of the dummies and creates (n-1) dummies only. Setting the 'prefix' argument helps to identify which columns are the dummy variables created for. It does this by adding whatever string you give the 'prefix' argument as a prefix to each of the dummy variables created. In our case 'rating_' will be appended to each dummy variable as a prefix.

```
dummy=pd.get_dummies(mydata['rating'],drop_first=True,prefix='rating')
```

The get_dummies() function has created a column for 'Excellent', 'Good' and 'Pathetic' but has dropped the column for 'Bad'.

```
dummy.head()
```

	rating_Excellent	rating_Good	rating_Pathetic
0	1	0	0
1	0	0	0
2	0	0	1
3	0	0	0
4	0	0	1

We can now simply attach these columns to the data using pandas concat() function.

```
mydata=pd.concat([mydata,dummy],axis=1)
```

The concat() function will take the axis argument as 1 since we are attaching the columns.

After creating dummies for 'rating', we will now drop the original column.

```
del mydata['rating']
```

```
mydata.columns
```

```
Index(['ID', 'age', 'balance', 'children', 'const_var', 'balance_log',
       'age_children_ratio', 'rating_score', 'f1', 'f2', 'city_mumbai',
       'city_chennai', 'city_kolkata', 'rating_Excellent', 'rating_Good',
       'rating_Pathetic'],
      dtype='object')
```

We need to keep in mind that we will not be doing all of what we learned here at once for any one dataset. Some of these techniques will be useful at a time while preparing data for machine learning algorithms.

Sorting and Merging Dataframes

Here will will cover the following:

1. Sorting data by one or more column(s) in ascending or descending manner
2. Combining dataframes vertically or horizontally
3. Merging dataframes using keys as well as left, right, inner and outer joins

Lets start with importing pandas and numpy.

```
import numpy as np
import pandas as pd
```

We will create a dataframe with random values.

```
df = pd.DataFrame(np.random.randint(2, 8,(20,4)), columns=['A','B','C','D'])
```

The random.randint() function creates 4 columns having 20 rows with values ranging between 2 and 8.

```
df.head()
```

	A	B	C	D
0	4	7	4	6
1	2	7	5	6
2	2	3	5	7
3	4	6	2	4
4	6	5	4	4

1. Sorting data by one or more column(s) in ascending or descending manner

If we wish to sort the dataframe by column A, we can do that using the sort_values() function on the dataframe.

```
df.sort_values("A").head()
```

	A	B	C	D
9	2	5	6	7
14	2	6	4	6
18	2	7	7	3
6	2	4	2	5
19	2	6	4	6

The output that we get is sorted by the column 'A'. But when we type 'df' again to view the dataframe, we see that there are not changes; df is the same as it was before sorting.

```
df.head()
```

	A	B	C	D
0	4	7	4	6
1	2	7	5	6
2	2	3	5	7
3	4	6	2	4
4	6	5	4	4

To handle this, there are two things that we can do:

- assign the sorted dataframe to the original dataframe
- use 'inplace=True' argument when sorting using sort_values() function

```
df = df.sort_values("A") # assign the sorted dataframe to the original dataframe
```

```
df.sort_values("A", inplace=True) # use 'inplace=True' argument when sorting using sort_values() function
```

Now when we observe the dataframe 'df', it will be sorted by 'A' in an ascending manner.

```
df.head()
```

	A	B	C	D
9	2	5	6	7
14	2	6	4	6
18	2	7	7	3
6	2	4	2	5
19	2	6	4	6

In case we wish to sort the dataframe in a descending manner, we can set the argument ascending=False in the sort_values() function.

```
df.sort_values("A", inplace=True, ascending=False)
```

Now the dataset will be sorted in the reverse order of the values of 'A'.

```
df.head()
```

	A	B	C	D
17	7	7	5	5
10	7	7	4	4
15	7	4	6	7
4	6	5	4	4
11	6	3	7	2

We can sort the dataframes by multiple columns also.

Sorting by next column in the sequence happens within the groups formed after sorting of the previous columns.

In the code below, we can see that the 'ascending' argument takes values [True, False]. It is passed in the same order as the columns ['B','C']. This means that the column 'B' will be sorted in an ascending order and within the groups created by column 'B', column 'C' will be sorted in a descending order.

```
df.sort_values(['B','C'],ascending=[True,False]).head(10)
```

	A	B	C	D
12	5	2	3	5
11	6	3	7	2
13	3	3	7	6
2	2	3	5	7
7	5	3	3	4
15	7	4	6	7
16	4	4	4	2
6	2	4	2	5
9	2	5	6	7
4	6	5	4	4

We can observe that the column 'B' is sorted in an ascending order. Within the groups formed by column 'B', column 'C' sorts its values in descending order.

Although we have not taken an explicit example for character data, in case of characater data, sorting happens in the order in which it is present in the dictionary.

3. Combining dataframes vertically or horizontally

Now we will see how to combine dataframes horizontally or vertically by stacking them.

Lets start by creating two dataframes.

```
df1 = pd.DataFrame([['a', 1], ['b', 2]],columns=['letter', 'number'])
df2 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],columns=['letter', 'number', 'animal'])
```

```
df1
```

	letter	number
0	a	1
1	b	2

```
df2
```

	letter	number	animal
0	c	3	cat
1	d	4	dog

```
import warnings
warnings.simplefilter(action='ignore', category=Warning) # This code is to ignore the warnings given by
Pandas
```

In order to combine these dataframes, we will use the concat() function of the pandas library.

The argument 'axis=0' combines the dataframes row-wise i.e. stacks the dataframes vertically.

```
pd.concat([df1,df2], axis=0)
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	animal	letter	number
0	NaN	a	1
1	NaN	b	2
0	cat	c	3
1	dog	d	4

We observe that since the column 'animal' is not present in 'df1' dataframe, the corresponding values in the concatenated dataframe have the values NaNs.

Notice that the index of the two dataframes is not generated afresh in the concatenated dataframe. The original indices are stacked, so we end up with duplicate index names. More often than not, we would not want the indices to be stacked. We can avoid doing this by setting the 'ignore_index' argument to True in the concat() function.

```
pd.concat([df1,df2], axis=0, ignore_index=True)
```

	animal	letter	number
0	NaN	a	1
1	NaN	b	2
2	cat	c	3
3	dog	d	4

We can see that the index is generated afresh.

We discussed how the dataframes can be stacked vertically. Now lets see how they can be stacked horizontally.

Lets create one more dataframe.

```
df3 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george'],['tiger','john']],columns=['animal', 'name'])
```

In order to stack the dataframes column-wise i.e. horizontally, we will need to set the 'axis' argument to 1.

The datasets which we will stack horizontally are 'df1' and 'df3'.

```
df1
```

	letter	number
0	a	1
1	b	2

```
df3
```

	animal	name
0	bird	polly
1	monkey	george
2	tiger	john

```
pd.concat([df1,df3],axis=1)
```

	letter	number	animal	name
0	a	1.0	bird	polly
1	b	2.0	monkey	george
2	NaN	NaN	tiger	john

We see that when we use the concat() function with 'axis=1' argument, we combine the dataframes column-wise.

Since 'df3' dataframe has three rows, whereas 'df1' dataframe has only two rows, the remaining values are set to missing as can be observed in the dataframe above.

Many times our datasets need to be combined by keys instead of simply stacking them vertically or horizontally. As an example, lets consider the following dataframes:

```
df1=pd.DataFrame({"custid":[1,2,3,4,5],
                  "product":["Radio","Radio","Fridge","Fridge","Phone"]})
df2=pd.DataFrame({"custid":[3,4,5,6,7],
                  "state":["UP","UP","UP","MH","MH"]})
```

```
df1
```

	custid	product
0	1	Radio
1	2	Radio
2	3	Fridge
3	4	Fridge
4	5	Phone

```
df2
```

	custid	state
0	3	UP
1	4	UP
2	5	UP
3	6	MH
4	7	MH

Dataframe 'df1' contains information about the customer id and the product they purchased and dataframe 'df2' also contains the customer id along with which state they come from.

Notice that the first row of the two dataframes have different customer ids i.e. the first row contains information about different customers, hence it won't make sense to combine the two dataframes together horizonatally.

In order to combine data from the two dataframes, we will first need to set a correspondence using customer id i.e. combine only those rows having a matching customer id and ignore the rest. In some situations, if there is data in one dataframe which is not present in the other dataframe, missing data will be filled in.

There are 4 ways in which the dataframes can be merged - inner join, outer join, left join and right join:

In the following code, we are joining the two dataframes 'df1' and 'df2' and the key or correspondence between the two dataframes is determined by 'custid' i.e. customer id. We use the inner join here (how='inner'), which retains only those rows which are present in both the dataframes. Since customer id's 3, 4 and 5 are common in both the dataframes, these three rows are returned as a result of the inner join along with corresponding information 'product' and 'state' from both the dataframes.

```
pd.merge(df1,df2,on=['custid'],how='inner')
```

	custid	product	state
0	3	Fridge	UP
1	4	Fridge	UP
2	5	Phone	UP

Now lets consider outer join. In outer join, we keep all the ids, starting at 1 and going up till 7. This leads to having missing values in some columns e.g. customer ids 6 and 7 were not present in the dataframe 'df1' containing product information. Naturally the product information for those customer ids will be absent.

Similarly, customer ids 1 and 2 were not present in the dataframe 'df2' containing state information. Hence, state information was missing for those customer ids.

Merging cannot fill in the data on its own if that information is not present in the original dataframes. We will explicitly see a lot of missing values in outer join.

```
pd.merge(df1,df2,on=['custid'],how='outer')
```

	custid	product	state
0	1	Radio	NaN
1	2	Radio	NaN
2	3	Fridge	UP
3	4	Fridge	UP
4	5	Phone	UP
5	6	NaN	MH
6	7	NaN	MH

Using the left join, we will see all customer ids present in the left dataframe 'df1' and only the corresponding product and state information from the two dataframes. The information present only in the right dataframe 'df2' is ignored i.e. customer ids 6 and 7 are ignored.

```
pd.merge(df1,df2,on=['custid'],how='left')
```

	custid	product	state
0	1	Radio	NaN
1	2	Radio	NaN
2	3	Fridge	UP
3	4	Fridge	UP
4	5	Phone	UP

Similarly, right join will contain all customer ids present in the right dataframe 'df2' irrespective of whether they are there in the left dataframe 'df1' or not.

```
pd.merge(df1,df2,on=['custid'],how='right')
```

	custid	product	state
0	3	Fridge	UP
1	4	Fridge	UP
2	5	Phone	UP
3	6	NaN	MH
4	7	NaN	MH

Numeric and Visual Summary of data

Here we will focus on how to look at summary of our data. We will observe how each variable in the data behaves individually and with other variables. We will understand how to get these summary statistics in Python.

We start with importing the numpy and pandas libraries.

```
import pandas as pd
import numpy as np
```

We will make use of bank-full.csv file available on LMS.

```
file=r'/Users/anjal/Dropbox/PDS V3/Data/bank-full.csv'
```

```
bd=pd.read_csv(file,delimiter=';')
```

Numeric summary of data

The first function we learn about is called describe(). It gives us a numerical summary of numeric variables. It returns 8 summary statistics i.e. count of non-missing values, mean, standard deviation, minimum, 25 percentile value, 50 percentile value, 75 percentile value and the maximum value. In order to understand the percentile values, lets consider the example of the variable 'balance'. For 'balance' 25% of the values are lower than 72. Median or the 50th percentile is 448. 75% of the values are either equal to or less than 1428; in other words, 25% or the values are greater than 1428.

```
bd.describe()
```

	age	balance	day	duration	campaign	pdays	previous
count	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000
mean	40.936210	1362.272058	15.806419	258.163080	2.763841	40.197828	0.580323
std	10.618762	3044.765829	8.322476	257.527812	3.098021	100.128746	2.303441
min	18.000000	-8019.000000	1.000000	0.000000	1.000000	-1.000000	0.000000
25%	33.000000	72.000000	8.000000	103.000000	1.000000	-1.000000	0.000000
50%	39.000000	448.000000	16.000000	180.000000	2.000000	-1.000000	0.000000
75%	48.000000	1428.000000	21.000000	319.000000	3.000000	-1.000000	0.000000
max	95.000000	102127.000000	31.000000	4918.000000	63.000000	871.000000	275.000000

Another useful function that can be applied on the entire data is nunique(). It returns the number of unique values taken by different variables.

Numeric data

```
bd.nunique()
```

```
age            77
job            12
marital         3
education       4
default         2
balance      7168
housing         2
loan            2
contact         3
day            31
month          12
duration     1573
campaign       48
pdays         559
previous       41
poutcome        4
y               2
dtype: int64
```

We can observe that the variables having the 'object' type have fewer values and variables which are 'numeric' have higher number of unique values.

The describe() function can be used with individual columns also. For numeric variables, it gives the 8 summary statistics for that column only.

```
bd['age'].describe()
```

```
count    45211.000000
mean        40.936210
std         10.618762
min         18.000000
25%         33.000000
50%         39.000000
75%         48.000000
max         95.000000
Name: age, dtype: float64
```

When the describe() function is used with a categorical column, it gives the total number of values in that column, total number of unique values, the most frequent value ('blue-collar') as well as the frequency of the most frequent value.

```
bd['job'].describe()
```

```
count           45211
unique             12
top       blue-collar
freq             9732
Name: job, dtype: object
```

Note: When we use the describe() function on the entire dataset, by default, it returns the summary statistics of numeric columns only.

There are tonnes of different summary statistics available.

Lets say we only wanted the mean or the median of the variable 'age'.

```
bd['age'].mean(), bd['age'].median()
```

```
(40.93621021432837, 39.0)
```

Apart from the summary statistics provided by the describe function, there are many other statistics available as shown below:

Function	Description
count	Number of non-null observations
sum	Sum of values
mean	Mean of values
mad	Mean absolute deviation
median	Arithmetic median of values
min	Minimum
max	Maximum
mode	Mode
abs	Absolute Value
prod	Product of values
std	Unbiased standard deviation
var	Unbiased variance
sem	Unbiased standard error of the mean
skew	Unbiased skewness (3rd moment)
kurt	Unbiased kurtosis (4th moment)
quantile	Sample quantile (value at %)
cumsum	Cumulative sum
cumprod	Cumulative product
cummax	Cumulative maximum
cummin	Cumulative minimum

Till now we have primarily discussed how to summarize numeric data.

Categorical data

Now starting with categorical data, we would want to look at frequency counts. We use the value_count() function for get the frequency counts of each unique element present in the column.

```
bd['job'].value_counts()
```

```
blue-collar      9732
management       9458
technician       7597
admin.           5171
services         4154
retired          2264
self-employed    1579
entrepreneur     1487
unemployed       1303
housemaid        1240
student           938
unknown           288
Name: job, dtype: int64
```

By default, the outcome of the value_counts() function is in descending order. The element 'blue-collar' with the highest count is displayed on the top and that with the lowest count 'unknown' is displayed at the bottom.

We should be aware of the format of the output. Lets save the outcome of the above code in a variable 'k'.

```
k = bd['job'].value_counts()
```

The outcome stored in 'k' has two attributes. One is values i.e. the raw frequencies and the other is 'index' i.e. the categories to which the frequencies belong.

```
k.values
```

```
array([9732, 9458, 7597, 5171, 4154, 2264, 1579, 1487, 1303, 1240,  938,
        288], dtype=int64)
```

```
k.index
```

```
Index(['blue-collar', 'management', 'technician', 'admin.', 'services',
       'retired', 'self-employed', 'entrepreneur', 'unemployed', 'housemaid',
       'student', 'unknown'],
      dtype='object')
```

As shown, values contain raw frequencies and index contains the corresponding categories. e.g. 'blue-collar' job has 9732 counts.

Lets say, you are asked to get the category with minimum count, you can directly get it with the following code:

```
k.index[-1]
```

```
'unknown'
```

We observe that the 'unknown' category has the least count.

We can get the category with the second highest count as well as the highest count as follows:

```
k.index[-2] # returns category with the second lowest count
```

```
'student'
```

```
k.index[0] # returns category with the highest count
```

```
'blue-collar'
```

Now if someone asks us for category names with frequencies higher than 1500. We can write the following code to get the same:

```
k.index[k.values>1500]
```

```
Index(['blue-collar', 'management', 'technician', 'admin.', 'services',
       'retired', 'self-employed'],
      dtype='object')
```

Even if we write the condition on k itself, by default it means that the condition is applied in the values.

```
k.index[k>1500]
```

```
Index(['blue-collar', 'management', 'technician', 'admin.', 'services',
       'retired', 'self-employed'],
      dtype='object')
```

The next kind of frequency table that we are interested in when working with categorical variables is cross-tabulation i.e. frequency of two categorical variables taken together. e.g. lets consider the cross-tabulation of two categorical variables 'default' and 'housing'.

```
pd.crosstab(bd['default'],bd['housing'])
```

housing	no	yes
default		
no	19701	24695
yes	380	435

In the above frequency table, we observe that there are 24695 observation where the value for 'housing' is 'yes' and 'default' is 'no'. This is a huge chunk of the population. There is a smaller chunk of about 435 observations where housing is 'yes' and default is 'yes' as well. Within the observations where default is 'yes', 'housing' is 'yes' for a higher numer of observations i.e. 435 as compared to where housing is 'no' i.e. 380.

Now, lets say that we want to look at the unique elements as well as the frequency counts of all categorical variables in the dataset 'bd'.

```
bd.select_dtypes('object').columns
```

```
Index(['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact',
       'month', 'poutcome', 'y'],
      dtype='object')
```

The code above will give us all the column names which are stored as categorical datatypes in the 'bd' dataframe. We can then run a for loop of top of these columns to get whichever summary statistic we need for the categorical columns.

```
cat_var = bd.select_dtypes('object').columns
for col in cat_var:
    print(bd[col].value_counts())
    print('~~~~~')
```

```
blue-collar      9732
management       9458
technician       7597
admin.           5171
services         4154
retired          2264
self-employed    1579
entrepreneur     1487
unemployed       1303
housemaid        1240
student           938
unknown           288
Name: job, dtype: int64
~~~~~
married     27214
single      12790
divorced     5207
Name: marital, dtype: int64
~~~~~
secondary    23202
tertiary     13301
primary       6851
unknown       1857
Name: education, dtype: int64
~~~~~
no     44396
yes      815
Name: default, dtype: int64
~~~~~
yes    25130
no     20081
Name: housing, dtype: int64
~~~~~
no     37967
yes     7244
Name: loan, dtype: int64
```

```
~~~~~
cellular     29285
unknown      13020
telephone     2906
Name: contact, dtype: int64
~~~~~
may    13766
jul     6895
aug     6247
jun     5341
nov     3970
apr     2932
feb     2649
jan     1403
oct      738
sep      579
mar      477
dec      214
Name: month, dtype: int64
~~~~~
unknown    36959
failure     4901
other       1840
success     1511
Name: poutcome, dtype: int64
~~~~~
no     39922
yes     5289
Name: y, dtype: int64
~~~~~
```

Many times we do not only want the summary statistics of numeric or categorical variables individually; we may want a summary of numeric variables within the categories coming from a categorical variable. e.g. lets say we want the average age of the people who are defaulting their loan as opposed to people who are not defaulting. This is known as group wise summary.

```
bd.groupby(['default'])['age'].mean()
```

```
default
no     40.961934
yes    39.534969
Name: age, dtype: float64
```

The result above tells us that the defaulters have a slightly lower average age as compared to non-defaulters.

Looking at median will give us a better idea in case we have many outliers. We notice that the difference is not much.

```
bd.groupby(['default'])['age'].median()
```

```
default
no     39
yes    38
Name: age, dtype: int64
```

We can group by multiple variables as well. There is no limit on the number and type of variables we can group by. But generally, we group by categorical variables only.

Also, it is not necessary of give the name of the column for which we want the summary statistic. e.g. in the code above, we wanted the median of the column 'age'. It is not necessary to specify the column 'age'. When we do not specify the column age, then we get a median of all the numeric columns grouped by the variable 'default'.

```
bd.groupby(['default']).median()
```

	age	balance	day	duration	campaign	pdays	previous
default							
no	39	468	16	180	2	-1	0
yes	38	-7	17	172	2	-1	0

We can also group by multiple variables as follows:

```
bd.groupby(['default','loan']).median()
```

		age	balance	day	duration	campaign	pdays	previous
default	**loan**							
no	**no**	39.0	509.0	16.0	181.0	2.0	-1.0	0.0
	yes	39.0	284.0	17.0	175.0	2.0	-1.0	0.0
yes	**no**	38.0	-3.5	17.0	178.5	2.0	-1.0	0.0
	yes	39.0	-21.0	18.0	163.0	2.0	-1.0	0.0

Each row in the result above gives the 4 categories defined by the two categorical variables we have grouped by and each column give the median value for all the numerical variables for each group.

In short, when we do not give a variable to compute the summary statistic e.g. median, we get all the columns where median can be computed.

Now, lets say we do not want to find the median for all columns, but only for 'day' and 'balance' columns. We can do that as follows:

```
bd.groupby(['housing','default'])['balance','day'].median()
```

		balance	day
housing	**default**		
no	**no**	531	17
	yes	0	18
yes	**no**	425	15
	yes	-137	15

Visual summary of data

Here we will understand how to summarize data visually. We will use the 'seaborn' library for this. Its a high level package built using matplotlib which is the base python visualization library.

If we were to do the visualizations using matplotlib instead of seaborn, we would need to write a lot more code. Seaborn has functions which wrap up this code making simpler.

```
import seaborn as sns
%matplotlib inline
```

Note: "%matplotlib inline" is required only when we use the Jupyter notebook so that visualizations appear within the notebook itself. Other editors like Spyder or PyCharm do not need this line of code as part of our script.

What we will cover is primarily to visualize our data quickly which will help us build our machine learning models.

We will learn to visualize the following:

1. Single numeric variables - density plots, box plots
2. Numeric numeric variables - scatter plot, pairwise density plots
3. Faceting of the data (visualizing chunks of data broken on the basis of categorical variables)
4. Categorical variables - frequency bar plots
5. Faceting of categorical variables
6. Heatmaps

```
import pandas as pd
import numpy as np
```

```
file=r'/Users/anjal/Dropbox/PDS V3/Data/bank-full.csv'
```

```
bd=pd.read_csv(file,delimiter=';')
```

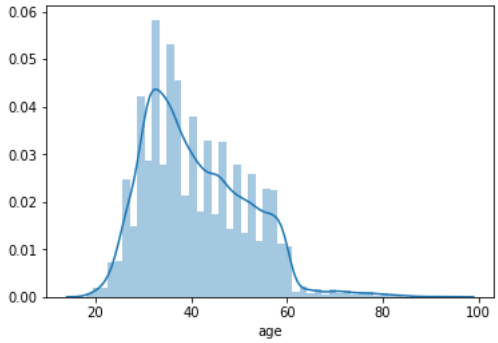
1. Visualizing single numeric variable

```
import warnings
warnings.simplefilter(action='ignore', category=Warning)
```

Density plots

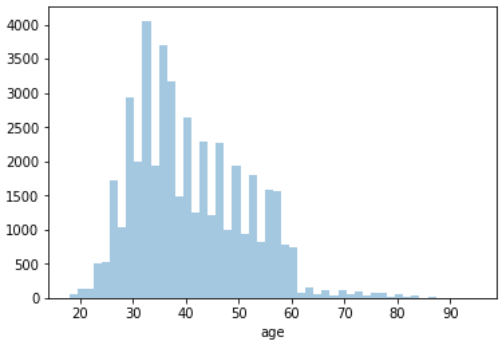
Lets start with density plots for a single numeric variable. We use the distplot() function from the seaborn library to get the density plot. The first agrument will be the variable for which we want to make the density plot.

```
sns.distplot(bd['age'])
```



By default, the distplot() function gives a histogram along with the density plot. In case we do not want the density plot, we can set the argument 'kde' (short for kernel density) to False.

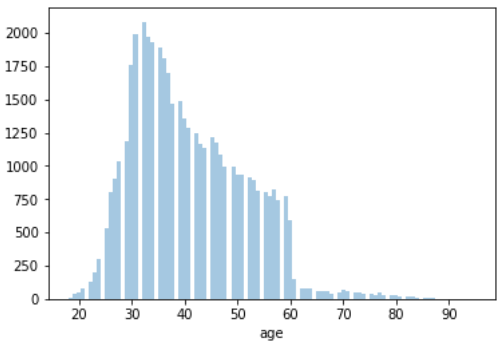
```
sns.distplot(bd['age'], kde=False)
```



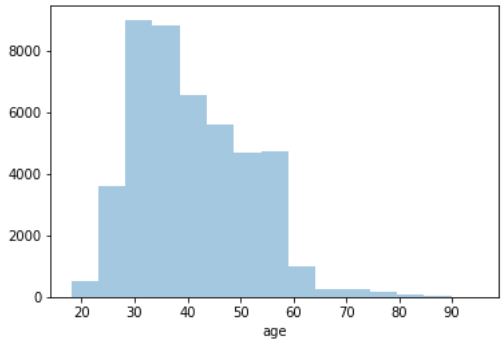
Setting the 'kde' argument to False will not show us the density curve, but will only show the histogram.

In order to build a histogram, continuous data is split into intervals called bins. The argument 'bins' lets us set the numer of bins which in turn affects the width of each bin. This argument has some default value, however we can increase the number of bins by changing the 'bin' argument.

```
sns.distplot(bd['age'], kde=False, bins=100)
```




```
sns.distplot(bd['age'], kde=False, bins=15)
```

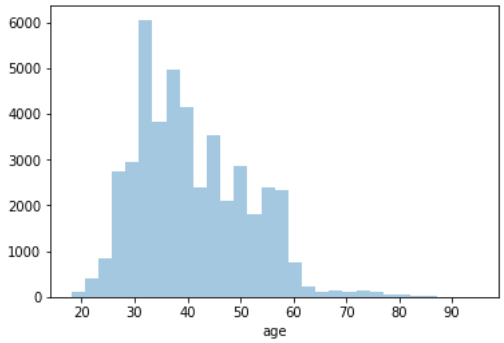


Notice the difference in the width of the bins when the argument 'bins' has different values. With 'bins' argument having the value 15, the bins are much wider as compared to when the 'bins' have the value 100.

How do we decide the number of bins, what would be a good choice? First consider why we need a histogram. Using a histogram we can get a fair idea where most of the values lie. e.g. considering the variable 'age', a histogram tells us how people in the data are disributed across different values of 'age' variable. Looking at the histogram, we get a fair idea that most of the people in our data lie in 30 to 40 age range. If we look a bit further, we can also say that the data primarily lies between 25 to about 58 years age range. Beyond the age of 58, the density falls. We might be looking at typical working age population.

Coming back to how do we decide the number of bins. Now, we can see that most of the people are between 30 to 40 years. Here, if we want to dig deeper to see how data is distributed within this range we increase the number of bins. In other words, when we want to go finer, we increase the number of bins.

```
sns.distplot(bd['age'], kde=False, bins=30)
```

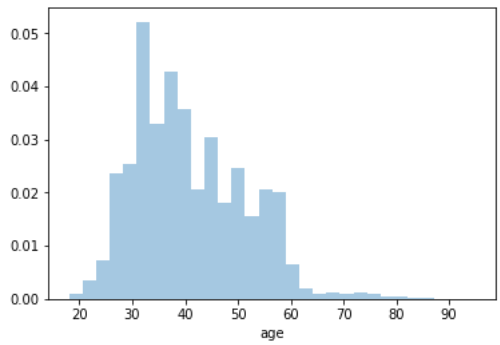


We can see here that between 30 to 40, the people in their early 30's are much more dominant as compared to the people whose age is closer to 40. One thing to be kept in mind when increasing the number of bins is that if the number of data points are very low, for example, if we have only 100 data points then it does not make sense to create 50 bins because the frequency bars that we see will not give us a very general picture.

In short, higher number of bins will give us a finer picture of how the data is behaving in terms of density across the value ranges. But with very few data points, a higher number of bins may give us a picture which may not be generalizable.

We can see that the y axis has frequencies. Sometimes it is much easier to look at frequency percentages. We get frequency percentages by setting the 'norm_hist' argument to True. It basically normalizes the histograms.

```
sns.distplot(bd['age'], kde=False, bins=30, norm_hist=True)
```

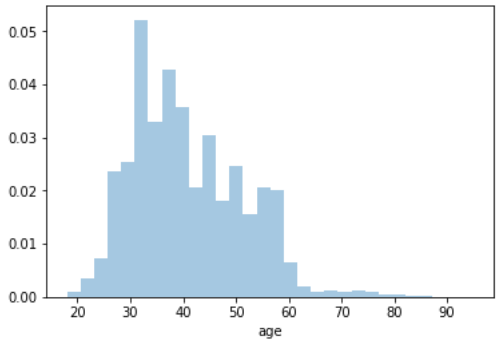


We can see that about 5% of the population lies between the age range of 31 to about 33.

Note: If we want to get more specific, we need to move towards numeric summary of data.

How do we save this graph as an image?

```
myplot = sns.distplot(bd['age'], kde=False, bins=30, norm_hist=True)
```



The code above saves the visualization in myplot.

```
myimage = myplot.get_figure()
```

myimage is something that we can save.

```
myimage.savefig("output.png")
```

The moment we do this, the 'output.png' file will appear wherever this script is. We can get this path using the pwd() function. My 'output.png' file is present in the follwing path:

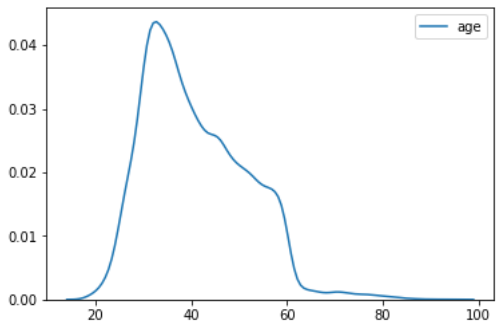
```
pwd()
```

```
'C:\\Users\\anjal\\Dropbox\\PDS V3\\2.Data_Prep'
```

The above method of saving images will work with all kinds of plots.

There is a function kdeplot() in seaborn that is used for generating only the density plot. It will only give us the density plot without the histogram.

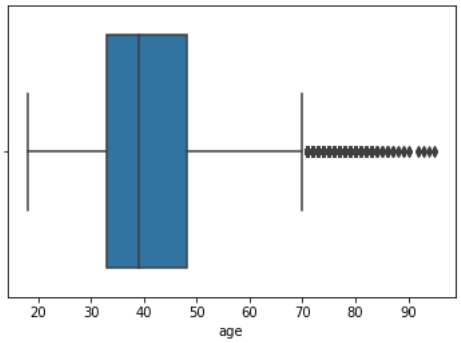
```
sns.kdeplot(bd['age'])
```



We observe that the different plots have their own functions and we simply need to pass the column which we wish to visualize.

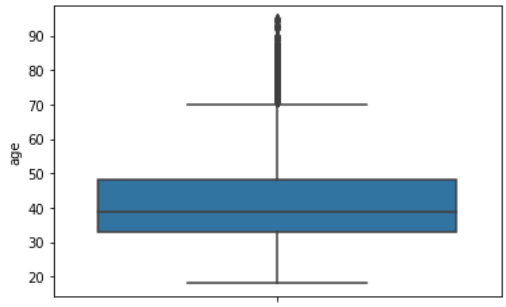
Now, let us see how to visualize the 'age' column using a boxplot.

```
sns.boxplot(bd['age'])
```



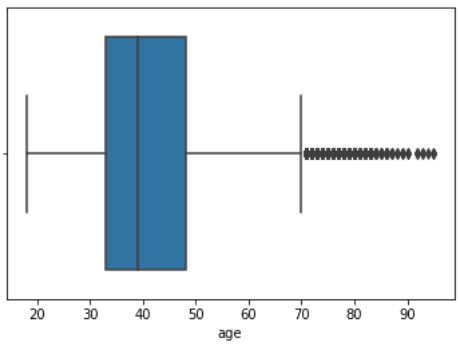
We can also get the above plot with the following code:

```
sns.boxplot(y='age', data=bd)
```



We get a vertical boxplot since we mentioned y as 'age'. We can also get the horizontal boxplot if we specify x as 'age'.

```
sns.boxplot(x='age', data=bd)
```

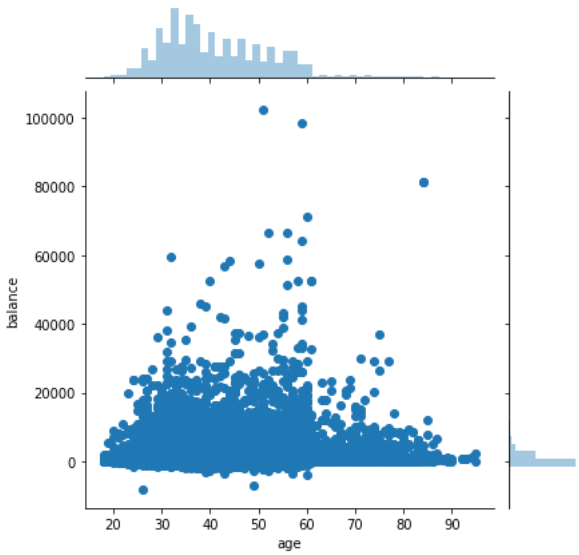


We can see that there are no extreme values on lower side of the age; however, there are a lot of extreme values on the higher side of age.

Visualizing numeric-numeric variables

We can use scatterplots to visualize two numeric columns together. The function which helps us do this is jointplot() from the seaborn library.

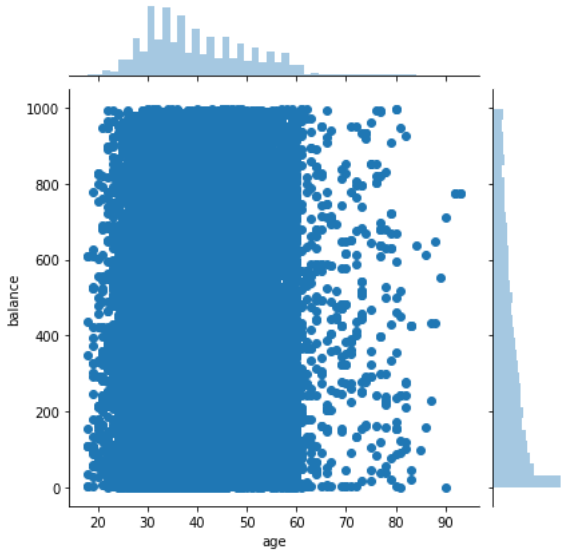
```
sns.jointplot(x='age', y='balance', data=bd)
```



The jointplot not only gives us the scatterplot but also gives the density plots along the axis.

We can do a lot more things with the jointplot() function. We observe that the variable 'balance' takes a value on a very long range but most of the values are concentrated on a very narrow range. Hence we will plot the data only for those observations for which 'balance' column has values ranging from 0 to 1000.

```
sns.jointplot(x="age", y="balance", data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:])
```

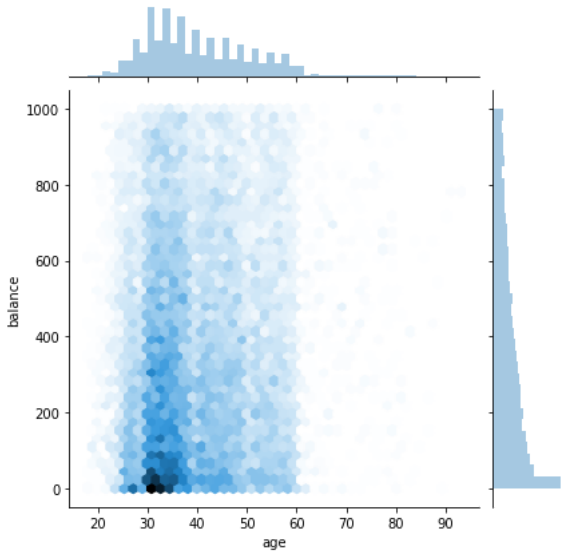


Note: Putting conditions is not a requirement for us to visualize the data. Since we see that most of the data lies in a smaller range and because of the few extreme values we may get a distorted plot.

Using the above code we have no way of figuring out if individual points are overlapping each other.

Setting the argument 'kind' as 'hex' not only shows us the observations but also helps us in knowing how many observations are overlapping at a point by observing the shade of each point. The darker the points, more the number of observations present there.

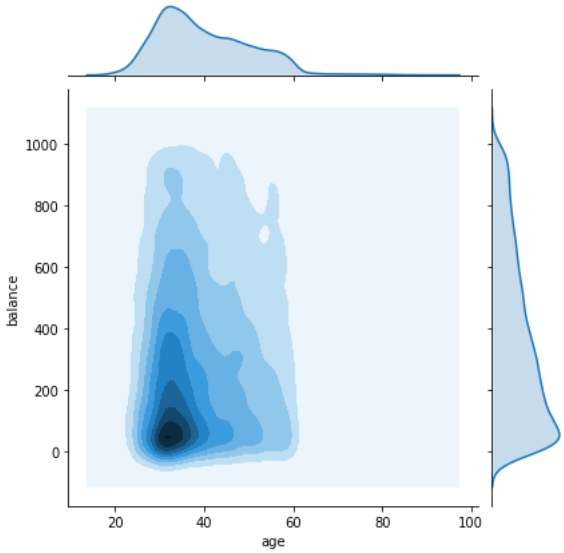
```
sns.jointplot(x="age", y="balance", data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:], kind='hex')
```



We can observe that most of the observations lie between the age of 30 and 40 and lot of observations lie between the balance of 0 to 400. As we move away, the shade keeps getting lighter indicating that the number of observations reduce or the density decreases.

These hex plots are a combination of scatterplots and pure density plots. If we want pure density plots, we need to change the 'kind' argument to 'kde'.

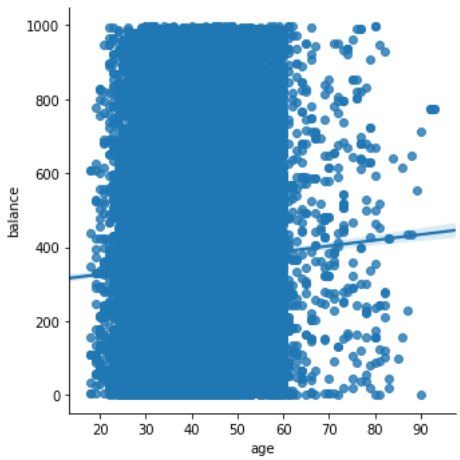
```
sns.jointplot(x="age", y="balance", data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:], kind='kde')
```



The darker shade indicates that most of the data lies there and as we move away the density of the data dissipates.

Next we will see the lmplot() function.

```
sns.lmplot(x='age', y='balance', data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:])
```

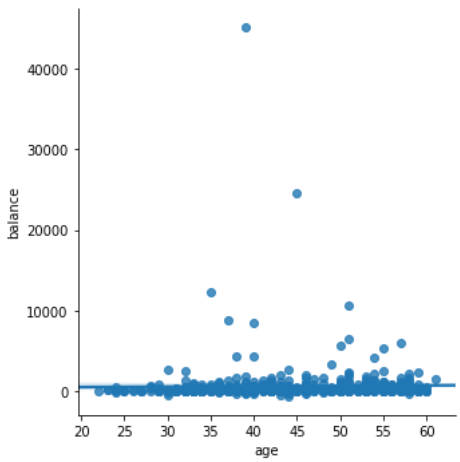


We observe that lmplot is just like scatter plot, but it fits a line through the data by default. (lmplot - linear model plot)

We can update lmplot to fit higher order polynomials which gives a sense if there exists a non-linear relationship between the data.

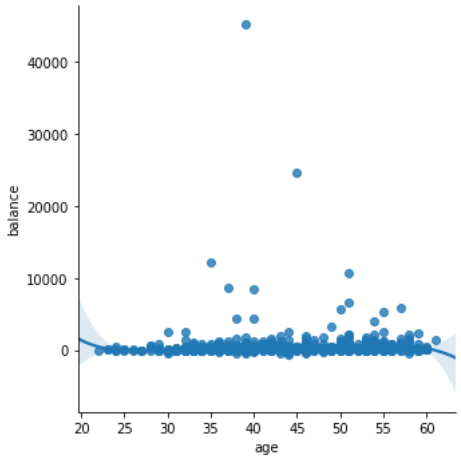
Since the data in the plot above is overlapping a lot, let us consider the first 500 observations only. We can see that a line fits through these points.

```
sns.lmplot(x='age', y='balance', data=bd.iloc[:500,:])
```



If we update the code above and add an argument 'order=6' we can see that the function has tried to fit a curve through the data. Since it still mostly looks like a line, so maybe there is a linear trend. Also, as the line is horizontal to the x axis, there is not much correlation between the two variables plotted.

```
sns.lmplot(x='age', y='balance', data=bd.iloc[:500,:], order=6)
```

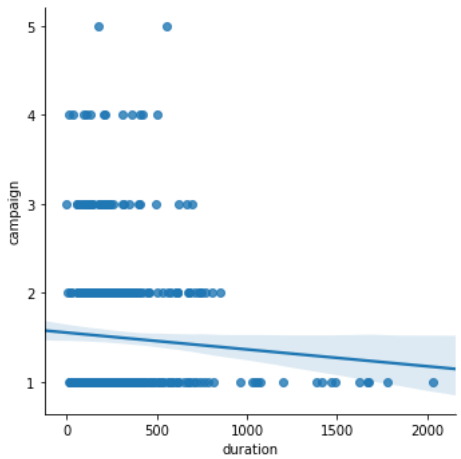


3. Faceting the data

As of now, we are looking at the age and balance relationship across the entire data.

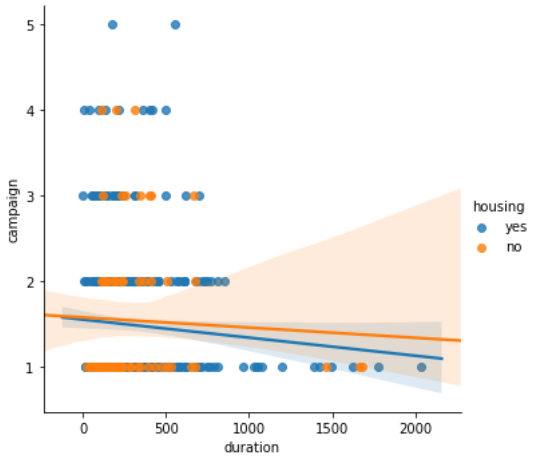
Let us see the relationship between 'duration' and 'campaign' variables.

```
sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:])
```



Now we want to see how will the relationship between 'duration' and 'campaign' behave for different values of 'housing'. We can observe this by coloring the datapoints for different values of 'housing' using the 'hue' argument. 'housing' variable takes two values: yes and no.

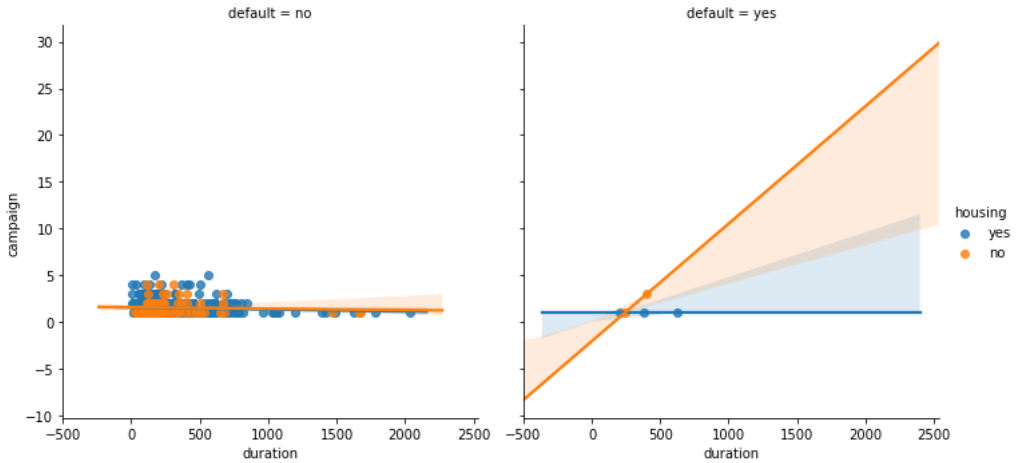
```
sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:], hue='housing')
```



We can see two different fitted lines. The orange one corresponds to 'housing' equal to no and the blue one corresponding to 'housing' equal to yes.

Now if we wish to divide our data further on the basis of 'default' column, we can consider using the 'col' argument. 'default' argument takes two values: yes and no.

```
sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:], hue='housing', col='default')
```

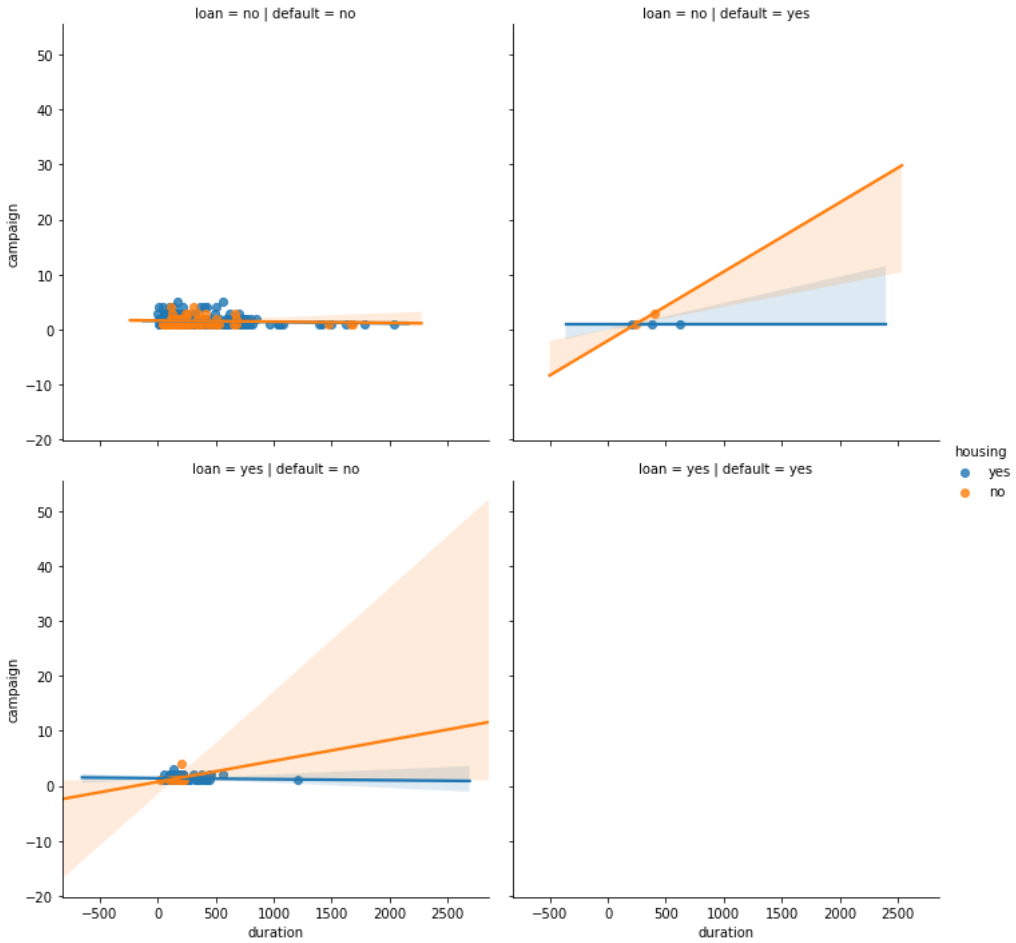


Now we have 4 parts of the data. Two are given by the color of the points and two more are given by separate columns. The first column refers to 'default' being no and the second column refers to 'default' being yes.

Observe that there are very few points when 'default' is equal to yes; hence we cannot trust the relationship as the data is very less.

Next, if we wanted to check how does the relationship between the two variables change when we are looking at different categories for loan. 'loan' also has two values: yes and no.

```
sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:], hue='housing', col='default', row='loan')
```



We observe, that within the group where 'default' is no; the relationship between the two variables 'campaign' and 'duration' is different when 'loan' is yes as compared to when 'loan' is no. Also, majority of the data points are present where 'loan' and 'default' both are no. There are no data points where both 'loan' and 'default' is yes. Keep in mind that we are looking at the first 500 observations only. We may get some data points here if we look at higher number of observations.

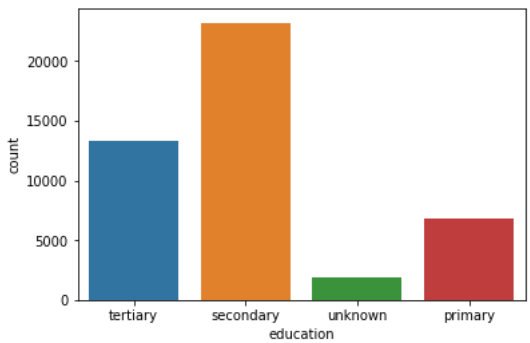
This how we can facet the data observing whether after breaking the data does the relationship between two variables change.

4. Visualizing categorical data

Lets start by making simple frequency bar charts using count plots.

We want to know how different education groups are present in the data.

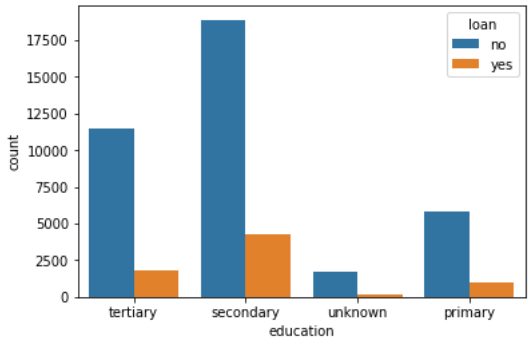
```
sns.countplot(x = 'education', data = bd)
```



We observe that the 'secondary' education group is very frequent. There is a small chunk where the level of education is 'unknown'. The 'tertiary' education group has the second highest count followed by 'primary'.

We have options to use faceting here as well. We can use the same syntax we used earlier. Lets start by adding 'hue' as 'loan'.

```
sns.countplot(x='education', data=bd, hue='loan')
```



We observe that each education level is now broken into two parts according to the value of 'loan'.

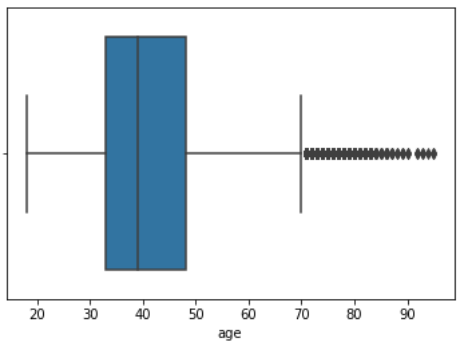
This is how we can facet using the same options for categorical data.

When we want to see how 'age' behaves across different levels of education, we can use boxplots().

When we make a boxplot only with the 'age' variable we get the following plot, indicating that the data primarily lies between age 20 and 70 with a few outlying values and the data overall is positively skewed.

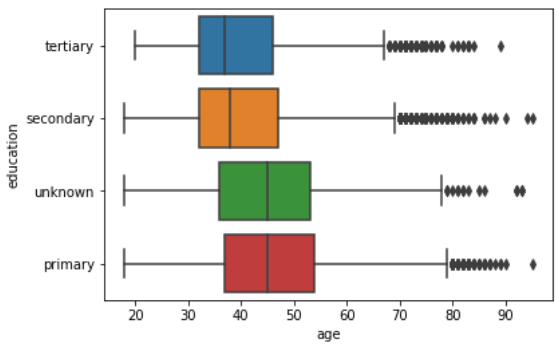
5. Faceting of categorical variables

```
sns.boxplot(x='age', data=bd)
```



Now, we want to look how the variable 'age' behaves across different levels of 'education'.

```
sns.boxplot(x='age', y='education', data=bd)
```



We observe that 'primary' education have overall higher median range. The behaviour of the data points under 'unknown' are similar to those under 'primary' education indicating maybe that the 'unknown' ones may have primary education background.

We also observe that people having 'tertiary' education belong to a lower age group. We can infer that older people could make do with lesser education but the current generation needs higher levels of education to get by. This could be one of the inferences.

6. Heatmaps

Heatmaps are two dimensional representation of data in which values are represented using colors. It uses a warm-to-cool color specturm to describe the values visually.

In order to understand heatmaps, lets start with generating a random array.

```
x = np.random.random(size=(20,20))
```

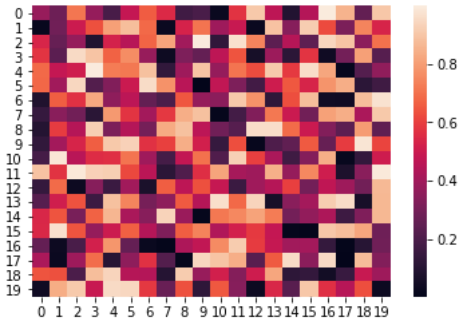
```
x[:3]
```

```
array([[0.37666211, 0.28033254, 0.71360714, 0.37308941, 0.19125123,
        0.48272533, 0.65697561, 0.54930338, 0.17009858, 0.19373981,
        0.02635841, 0.57243852, 0.90673713, 0.4668148 , 0.09514138,
        0.44202699, 0.99756982, 0.85043029, 0.25563275, 0.90301468],
       [0.02407724, 0.2872011 , 0.5026282 , 0.63615388, 0.80489701,
        0.88052132, 0.68300948, 0.0335487 , 0.52819662, 0.70863449,
        0.39304293, 0.48927019, 0.02993629, 0.89080078, 0.33833946,
        0.91885461, 0.62108977, 0.31585145, 0.74250102, 0.5337096 ],
       [0.53687923, 0.26465329, 0.37573166, 0.09014275, 0.52514711,
        0.45017881, 0.67338532, 0.81795034, 0.39931306, 0.99530252,
        0.12930078, 0.96921   , 0.74296808, 0.24080314, 0.44930359,
        0.24761788, 0.94031158, 0.89358538, 0.35454129, 0.7008932 ]])
```

Looking at the data above, it is difficult for us to determine what kind of values it has.

If we pass this data to the heatmap() function, it will be displayed as below:

```
sns.heatmap(x)
```



When we observe a heatmap, wherever the color of the boxes are light, the values are closer to 1 and as the boxes get darker, those are the values closer to 0. Looking at the visualization above, we get an idea that more or less the values are quite random. There does not seems to be any dominance of lighter or darker boxes.

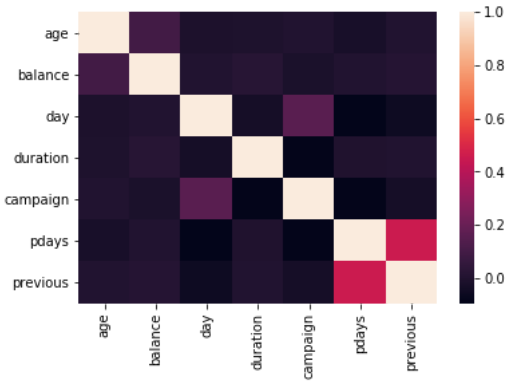
Now, since we understand the use of colors in heatmaps, lets get back to understanding how does it help with understanding our 'bd' dataset. Lets say we look at the correlations in the data 'bd'.

```
bd.corr()
```

	age	balance	day	duration	campaign	pdays	previous
age	1.000000	0.097783	-0.009120	-0.004648	0.004760	-0.023758	0.001288
balance	0.097783	1.000000	0.004503	0.021560	-0.014578	0.003435	0.016674
day	-0.009120	0.004503	1.000000	-0.030206	0.162490	-0.093044	-0.051710
duration	-0.004648	0.021560	-0.030206	1.000000	-0.084570	-0.001565	0.001203
campaign	0.004760	-0.014578	0.162490	-0.084570	1.000000	-0.088628	-0.032855
pdays	-0.023758	0.003435	-0.093044	-0.001565	-0.088628	1.000000	0.454820
previous	0.001288	0.016674	-0.051710	0.001203	-0.032855	0.454820	1.000000

Normally the correlation tables can be quite huge, can have 50 variables in the data too. Looking at this table, it is very difficult to manually check if there exists correlation between the variables. We can manually check if any of the values in the table above are close to 1 or -1 which indicates high correlation or we can simply pass the above table to a heatmap.

```
sns.heatmap(bd.corr())
```



The visualization above shows that wherever the boxes are very light i.e. near +1, there is high correlation and wherever the boxes are too dark i.e. near 0, the correlation is low. The diagonal will be the lightest because each variable has maximum correlation with itself. But for the rest of the data, we observe that there is not much correlation. However, there seems to be some positive correlation between 'previous' and 'pdays'. The advantage of using heatmaps is that we do not have to go through the correlation table to manually check if correlation is present or not. We can visually understand the same through the heatmap shown above.