Pandas

What will you learn?

1. Introduction to Pandas
2. Reading the Data
3. **Functionalities of Pandas** : Creation, Viewing, Editing
4. Manipulating Data
5. Handling NaN
6. **Handling Duplicates** : Row Index, Column Names
7. Handling String Data

Pandas is an open source library which provides high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Pandas has a lot of functions that will help in reading and writing data and also for data manipulation. Thus we will be using pandas throughout the course.

Pandas behave like an excel file.

Lets import pandas and read some data.

In [ ]:
```
#Import Pandas
import pandas as pd
```

Reading Data

We will use **read_csv()** function. It reads a comma-separated values (csv) file into DataFrame.

In [ ]:
```
#Loading data with read_csv() function. Here we are providing path to the csv file
#If the file is in your system you can provide its path as well.
iris = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/
```

In [ ]:
```
type(iris)
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

Pandas Dataframes

DataFrame is an object for data manipulation. You can think of it as a 2D tabular structure, where every row is a dataset entry and columns represents features of data.

In [ ]:
```
iris
```

Out[ ]:

	5.1	3.5	1.4	0.2	Iris-setosa
0	4.9	3.0	1.4	0.2	Iris-setosa
1	4.7	3.2	1.3	0.2	Iris-setosa
2	4.6	3.1	1.5	0.2	Iris-setosa
3	5.0	3.6	1.4	0.2	Iris-setosa
4	5.4	3.9	1.7	0.4	Iris-setosa
...	...	...	...	...	...
144	6.7	3.0	5.2	2.3	Iris-virginica
145	6.3	2.5	5.0	1.9	Iris-virginica
146	6.5	3.0	5.2	2.0	Iris-virginica
147	6.2	3.4	5.4	2.3	Iris-virginica
148	5.9	3.0	5.1	1.8	Iris-virginica

149 rows × 5 columns

By default, the first row of the csv file has been used as column names. We will soon see how to fix that.

Creating copy of DataFrame

In [ ]:

```
df = iris
## Above statement simply makes df refer to the data frame object that iris is refe
## So now both iris and df refer to the same dataframe object and any changes done
## So effectively this is not creating another dataframe object.
```

If we wish to create a copy then we will use **copy()** function for that

In [ ]:

```
df = iris.copy()
```

In [ ]:

```
df.shape
```

Out[ ]:

```
(149, 5)
```

As you can see, we have 149 rows and 5 columns. But actually, this should have been 150 rows, as we already know, the Iris Dataset has information of 3 different types of flower, 50 each. This happened because the first row was taken as the column name. To fix this, we do the following:

In [ ]:

```
#Ignoring header -> If you don't want first row to be treated as a header, you can
iris = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/
iris
```

Out[ ]:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

In [ ]:
```
df = iris.copy()
df.shape
```

Out[ ]:
```
(150, 5)
```

To see the datatypes of each column we do the following:

In [ ]:
```
df.dtypes
```

Out[ ]:
```
0    float64
1    float64
2    float64
3    float64
4     object
dtype: object
```

Currently, our columns have no names.

In [ ]:
```
df.columns
```

Out[ ]:
```
Int64Index([0, 1, 2, 3, 4], dtype='int64')
```

To give them a name, we simply change the value of df.columns

In [ ]:
```
df.columns = ['sl', 'sw', 'pl', 'pw', 'flower_type']
df
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

In [ ]:
```
df.dtypes
```

Out[ ]:
```
sl              float64
sw              float64
pl              float64
pw              float64
flower_type      object
dtype: object
```

We may get a quick analysis of our data using **describe()**

In [ ]:
```
df.describe()
```

Out[ ]:

	sl	sw	pl	pw
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Some Basic Functionalties

Viewing the DataFrame

We have the **head()** and **tail()** function for viewing the dataframe.

head()

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

By default, value of n = 5.

In [ ]:
```
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [ ]:
```
df.head(10)
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

tail()

This function returns the last n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

By default, value of n = 5.

In [ ]:
```
df.tail()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

In [ ]:

```
df.tail(11)
```

Out[ ]:

	sl	sw	pl	pw	flower_type
139	6.9	3.1	5.4	2.1	Iris-virginica
140	6.7	3.1	5.6	2.4	Iris-virginica
141	6.9	3.1	5.1	2.3	Iris-virginica
142	5.8	2.7	5.1	1.9	Iris-virginica
143	6.8	3.2	5.9	2.3	Iris-virginica
144	6.7	3.3	5.7	2.5	Iris-virginica
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Accessing Data

Sometimes, we may want to look at a single column from the DataFrame. This can be done simply as:

In [ ]:

```
## Viewing sl column
df.sl
```

Out[ ]:

```
0      5.1
1      4.9
2      4.7
3      4.6
4      5.0
      ...
145    6.7
146    6.3
147    6.5
148    6.2
149    5.9
Name: sl, Length: 150, dtype: float64
```

and

In [ ]:

```
df['sl']
```

Out[ ]:
```
0      5.1
1      4.9
2      4.7
3      4.6
4      5.0
      ...
145    6.7
146    6.3
147    6.5
148    6.2
149    5.9
Name: sl, Length: 150, dtype: float64
```

Checking for NULL values

In [ ]:
```
df.isnull()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
...	...	...	...	...	...
145	False	False	False	False	False
146	False	False	False	False	False
147	False	False	False	False	False
148	False	False	False	False	False
149	False	False	False	False	False

150 rows × 5 columns

In [ ]:
```
# To get a direct overview
df.isnull().sum()
```

Out[ ]:
```
sl             0
sw             0
pl             0
pw             0
flower_type    0
dtype: int64
```

Selection

iloc[]

We can use the **iloc[ ]** function to access values in dataframe.

It is a purely integer-location based indexing for selection by position. iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

1. An integer, e.g. 5.
2. A list or array of integers, e.g. [4, 3, 0].
3. A slice object with ints, e.g. 1:7.
4. A boolean array.

In [ ]:
```
df.iloc[1:4, 2:4]
```

Out[ ]:

	pl	pw
1	1.4	0.2
2	1.3	0.2
3	1.5	0.2

loc[ ]

This accesses a group of rows and columns by label(s) or a boolean array.

.loc[ ] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

1. A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
2. A list or array of labels, e.g. ['a', 'b', 'c'].
3. A slice object with labels, e.g. 'a':'f'.
4. A boolean array of the same length as the axis being sliced, e.g. [True, False, True].

In [ ]:
```
df1 = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
     index=['cobra', 'viper', 'sidewinder'],
     columns=['max_speed', 'shield'])
df1
```

Out[ ]:

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

In [ ]:
```
df1.loc['viper']
```

Out[ ]:
```
max_speed    4
shield       5
Name: viper, dtype: int64
```

In [ ]:
```
df1.loc[['viper', 'sidewinder']]
```

Out[ ]:

	max_speed	shield
viper	4	5
sidewinder	7	8

DataFrame from Dictionary

```
In [ ]: mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
                  {'a': 100, 'b': 200, 'c': 300, 'd': 400},
                  {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
        df1 = pd.DataFrame(mydict)
        df1
```

Out[ ]:

	a	b	c	d
0	1	2	3	4
1	100	200	300	400
2	1000	2000	3000	4000

Manipulating data

Deletion of data

drop()

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

It returns us a DataFrame without the removed index or column labels, or None if inplace=True.

```
In [ ]: df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
In [ ]: a = df.drop(0)
        a.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa

To actually change the data in the original dataframe, we use the parameter 'inplace = True'

In [ ]:
```
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [ ]:
```
df.drop(0, inplace = True)
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa

Let's try to do this again

In [ ]:
```
df.drop(0, inplace = True)   #Error Generated
df.head()
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-32-215644c67776> in <module>()
----> 1 df.drop(0, inplace = True)   #Error Generated
      2 df.head()

/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py in drop(self, labels,
 axis, index, columns, level, inplace, errors)
   4172             level=level,
   4173             inplace=inplace,
-> 4174             errors=errors,
   4175         )
   4176 

/usr/local/lib/python3.6/dist-packages/pandas/core/generic.py in drop(self, label
s, axis, index, columns, level, inplace, errors)
   3887         for axis, labels in axes.items():
   3888             if labels is not None:
-> 3889                 obj = obj._drop_axis(labels, axis, level=level, errors=err
ors)
   3890 
   3891         if inplace:

/usr/local/lib/python3.6/dist-packages/pandas/core/generic.py in _drop_axis(self,
 labels, axis, level, errors)
   3921                 new_axis = axis.drop(labels, level=level, errors=errors)
   3922             else:
-> 3923                 new_axis = axis.drop(labels, errors=errors)
   3924             result = self.reindex(**{axis_name: new_axis})
   3925 

/usr/local/lib/python3.6/dist-packages/pandas/core/indexes/base.py in drop(self, l
abels, errors)
   5285         if mask.any():
   5286             if errors != "ignore":
-> 5287                 raise KeyError(f"{labels[mask]} not found in axis")
   5288             indexer = indexer[~mask]
   5289         return self.delete(indexer)

KeyError: '[0] not found in axis'
```

The reason for this is, after dropping 0, the indexing did not change automatically. Now, the labels do not begin from 0, but 1.

As we learnt in the definition, we are removing rows by their labels. To remove rows by their indices, we may do the following:

In [ ]:
```
df.drop(df.index[0], inplace = True)
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa

In [ ]:
```
df.drop(df.index[3], inplace = True)    ## Label 5 removed
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa

We may also remove many labels in one go.

In [ ]:
```
df.drop(df.index[[3, 4]], inplace = True)    ## Label 6, 7 removed
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

In a similar manner, we may remove columns.

In [ ]:
```
df.drop('sl')    ## Error Generated
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-36-396628fddc03> in <module>()
----> 1 df.drop('sl')    ## Error Generated

/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py in drop(self, labels, axis, index, columns, level, inplace, errors)
   4172             level=level,
   4173             inplace=inplace,
-> 4174             errors=errors,
   4175         )
   4176 

/usr/local/lib/python3.6/dist-packages/pandas/core/generic.py in drop(self, labels, axis, index, columns, level, inplace, errors)
   3887         for axis, labels in axes.items():
   3888             if labels is not None:
-> 3889                 obj = obj._drop_axis(labels, axis, level=level, errors=errors)
   3890 
   3891         if inplace:

/usr/local/lib/python3.6/dist-packages/pandas/core/generic.py in _drop_axis(self, labels, axis, level, errors)
   3921                 new_axis = axis.drop(labels, level=level, errors=errors)
   3922             else:
-> 3923                 new_axis = axis.drop(labels, errors=errors)
   3924             result = self.reindex(**{axis_name: new_axis})
   3925 

/usr/local/lib/python3.6/dist-packages/pandas/core/indexes/base.py in drop(self, labels, errors)
   5285         if mask.any():
   5286             if errors != "ignore":
-> 5287                 raise KeyError(f"{labels[mask]} not found in axis")
   5288             indexer = indexer[~mask]
   5289         return self.delete(indexer)

KeyError: "['sl'] not found in axis"
```

An error is generated because the drop function is currently looking for a row with label 'sl'. We need to change the axis.

In [ ]:
```
df.drop('sl', axis = 1)
```

Conditional Insights

We may use concept of boolean indexing in DataFrame to access a particular type of data, and draw inferenced from it.

In [ ]:
```
df
```

Out[ ]:

	sl	sw	pl	pw	flower_type
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

145 rows × 5 columns

Lets try to gain insights of data correspondign to Iris-virginica.

In [ ]:

```
df[df.flower_type == 'Iris-virginica'].describe()
```

Out[ ]:

	sl	sw	pl	pw
count	50.00000	50.000000	50.000000	50.00000
mean	6.58800	2.974000	5.552000	2.02600
std	0.63588	0.322497	0.551895	0.27465
min	4.90000	2.200000	4.500000	1.40000
25%	6.22500	2.800000	5.100000	1.80000
50%	6.50000	3.000000	5.550000	2.00000
75%	6.90000	3.175000	5.875000	2.30000
max	7.90000	3.800000	6.900000	2.50000

Addition of data

loc()

In [ ]:

```
df.loc[0] = [1, 2, 3, 4, 'Iris-virginica']
df.tail()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica
0	1.0	2.0	3.0	4.0	Iris-virginica

We may directly create new columns also according to our needs.

In [ ]:
```
df["diff_of_sl_sw"] = df['sl'] - df['sw']
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type	diff_of_sl_sw
2	4.7	3.2	1.3	0.2	Iris-setosa	1.5
3	4.6	3.1	1.5	0.2	Iris-setosa	1.5
4	5.0	3.6	1.4	0.2	Iris-setosa	1.4
8	4.4	2.9	1.4	0.2	Iris-setosa	1.5
9	4.9	3.1	1.5	0.1	Iris-setosa	1.8

In [ ]:
```
df.drop('diff_of_sl_sw', axis = 1, inplace = True)
```

Reset Index

After removing certain rows, the order of indices got changed. We can reset it using the **reset_index()** function.

In [ ]:
```
df.reset_index()
```

Out[ ]:

	index	sl	sw	pl	pw	flower_type
0	2	4.7	3.2	1.3	0.2	Iris-setosa
1	3	4.6	3.1	1.5	0.2	Iris-setosa
2	4	5.0	3.6	1.4	0.2	Iris-setosa
3	8	4.4	2.9	1.4	0.2	Iris-setosa
4	9	4.9	3.1	1.5	0.1	Iris-setosa
...	...	...	...	...	...	...
141	146	6.3	2.5	5.0	1.9	Iris-virginica
142	147	6.5	3.0	5.2	2.0	Iris-virginica
143	148	6.2	3.4	5.4	2.3	Iris-virginica
144	149	5.9	3.0	5.1	1.8	Iris-virginica
145	0	1.0	2.0	3.0	4.0	Iris-virginica

146 rows × 6 columns

But this has created an additional column with old indices. To avoid that, we do:

In [ ]:
```
df.reset_index(drop = True)
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	4.7	3.2	1.3	0.2	Iris-setosa
1	4.6	3.1	1.5	0.2	Iris-setosa
2	5.0	3.6	1.4	0.2	Iris-setosa
3	4.4	2.9	1.4	0.2	Iris-setosa
4	4.9	3.1	1.5	0.1	Iris-setosa
...	...	...	...	...	...
141	6.3	2.5	5.0	1.9	Iris-virginica
142	6.5	3.0	5.2	2.0	Iris-virginica
143	6.2	3.4	5.4	2.3	Iris-virginica
144	5.9	3.0	5.1	1.8	Iris-virginica
145	1.0	2.0	3.0	4.0	Iris-virginica

146 rows × 5 columns

Handling NaN

Values considered “missing”

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python None will arise and we wish to also consider that “missing” or “not available” or “NA”.

To make detecting missing values easier (and across different array dtypes), pandas provides the **isna()** and **notna()** functions, which are also methods on Series and DataFrame objects.

Because NaN is a float, a column of integers with even one missing values is cast to floating-point dtype

NaN values can create inaccuracies in our estimations and calculations. There are two ways we can handle NaN:

1. we either remove them,
2. or we fill them.

Our current data does not have any NaN values, so we will create some.

In [ ]:
```
import numpy as np
```

```
df = iris.copy()
df.columns = ['sl', 'sw', 'pl', 'pw', 'flower_type']
```

In [ ]:
```
df.iloc[2:4, 1:3] = np.nan
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	NaN	NaN	0.2	Iris-setosa
3	4.6	NaN	NaN	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [ ]:
```
df.describe()
```

Out[ ]:

	sl	sw	pl	pw
count	150.000000	148.000000	148.000000	150.000000
mean	5.843333	3.052703	3.790541	1.198667
std	0.828066	0.436349	1.754618	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.400000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Dropping NaN

dropna() : This will remove the row or column entries with NaN values.

In [ ]:
```
df.dropna(inplace = True)  ## Remove NaN inside df only
df.reset_index(drop = True, inplace = True)   ## Reset the indices
```

In [ ]:
```
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	5.0	3.6	1.4	0.2	Iris-setosa
3	5.4	3.9	1.7	0.4	Iris-setosa
4	4.6	3.4	1.4	0.3	Iris-setosa

As you may observe, we have removed the row with NaN. If we want to remove the column, we shall use 'axis' parameter.

Filling NaN

fillna() : You can also fill NaN using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill.

Generally we fill the NaN values with the mean, but depending on the type of data, and your own analysis, you may decide to will NaN in some other way.

In [ ]:
```
df.iloc[2:4, 1:3] = np.nan
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	5.0	NaN	NaN	0.2	Iris-setosa
3	5.4	NaN	NaN	0.4	Iris-setosa
4	4.6	3.4	1.4	0.3	Iris-setosa

In [ ]:
```
df.sw.fillna(df.sw.mean(), inplace = True)
df.pl.fillna(df.pl.mean(), inplace = True)
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.500000	1.400000	0.2	Iris-setosa
1	4.9	3.000000	1.400000	0.2	Iris-setosa
2	5.0	3.043151	3.821233	0.2	Iris-setosa
3	5.4	3.043151	3.821233	0.4	Iris-setosa
4	4.6	3.400000	1.400000	0.3	Iris-setosa

Note: Since all the NaN values belonged to 'Iris-setosa', a better value to fill NaN's would have been the mean of those values of 'sw', where flower type is Iris-setosa.

In [ ]:
```
df.iloc[2:4, 1:3] = np.nan
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	5.0	NaN	NaN	0.2	Iris-setosa
3	5.4	NaN	NaN	0.4	Iris-setosa
4	4.6	3.4	1.4	0.3	Iris-setosa

In [ ]:
```
df_setosa = df[df.flower_type == 'Iris-setosa']
df.sw.fillna(df_setosa.sw.mean(), inplace = True)
df.pl.fillna(df_setosa.pl.mean(), inplace = True)
df.head()
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.500000	1.400000	0.2	Iris-setosa
1	4.9	3.000000	1.400000	0.2	Iris-setosa
2	5.0	3.415217	1.463043	0.2	Iris-setosa
3	5.4	3.415217	1.463043	0.4	Iris-setosa
4	4.6	3.400000	1.400000	0.3	Iris-setosa

Duplicate Labels

Index objects are not required to be unique; you can have duplicate row or column labels.

But one of pandas' roles is to clean messy, real-world data before it goes to some downstream system. And real-world data has duplicates, even in fields that are supposed to be unique.

Lets see how duplicate labels change the behavior of certain operations, and how prevent duplicates from arising during operations, or to detect them if they do.

Consequences of Duplicate Labels

Some pandas methods (Series.reindex() for example) just don't work with duplicates present. The output can't be determined, and so pandas raises.

Other methods, like indexing, can give very surprising results. Typically indexing with a scalar will reduce dimensionality. Slicing a DataFrame with a scalar will return a Series. Slicing a Series with a scalar will return a scalar. But with duplicates, this isn't the case.

In [ ]:

```
df1 = pd.DataFrame([[0, 1, 2], [3, 4, 5]], columns=["A", "A", "B"])
df1
```

Out[ ]:

	A	A	B
0	0	1	2
1	3	4	5

We have duplicates in the columns. If we slice 'B', we get back a Series

In [ ]:

```
print(df1["B"])  # a series
type(df1["B"])
```

```
0    2
1    5
Name: B, dtype: int64
```

Out[ ]:

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

But slicing 'A' returns a DataFrame

In [ ]:

```
print(df1["A"]) # a DataFrame
type(df1["A"])
```

```
   A  A
0  0  1
1  3  4
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

This applies to row labels as well.

In [ ]:
```
df2 = pd.DataFrame({"A": [0, 1, 2]}, index=["a", "a", "b"])
df2
```

Out[ ]:

	A
a	0
a	1
b	2

In [ ]:
```
df2.loc["b", "A"]  # a scalar
```

Out[ ]:
```
2
```

In [ ]:
```
df2.loc["a", "A"]  # a Series
```

Out[ ]:
```
a    0
a    1
Name: A, dtype: int64
```

Duplicate Label Detection

You can check whether an Index (storing the row or column labels) is unique with **Index.is_unique**:

In [ ]:
```
df2
```

Out[ ]:

	A
a	0
a	1
b	2

In [ ]:
```
df2.index.is_unique
```

Out[ ]:
```
False
```

In [ ]:
```
df2.columns.is_unique
```

Out[ ]:
```
True
```

Index.duplicated() will return a boolean ndarray indicating whether a label is repeated.

In [ ]:
```
df2.index.duplicated()
```

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

Handling Strings in Data

Our algorithms can make calculations over numerical data. String data is very hard to compute quantitaviely.

It wont make sense to ignore string data. For example, if a dataset is to evaluate shopping habits, and we have a column for gender with categories as 'male' and 'female', we cannot just ignore this, as the habits of both the gender will be very different from each other.

So, to handle such cases, we convert the string data to numerical data.

In [ ]:
```
df
```

Out[ ]:

	sl	sw	pl	pw	flower_type
0	5.1	3.500000	1.400000	0.2	Iris-setosa
1	4.9	3.000000	1.400000	0.2	Iris-setosa
2	5.0	3.415217	1.463043	0.2	Iris-setosa
3	5.4	3.415217	1.463043	0.4	Iris-setosa
4	4.6	3.400000	1.400000	0.3	Iris-setosa
...	...	...	...	...	...
143	6.7	3.000000	5.200000	2.3	Iris-virginica
144	6.3	2.500000	5.000000	1.9	Iris-virginica
145	6.5	3.000000	5.200000	2.0	Iris-virginica
146	6.2	3.400000	5.400000	2.3	Iris-virginica
147	5.9	3.000000	5.100000	1.8	Iris-virginica

148 rows × 5 columns

Lets create a dummy column to understand the process.

In [ ]:
```
df['Gender'] = 'Female'
df.iloc[0:10, 5] = 'Male'
df
```

Out[ ]:

	sl	sw	pl	pw	flower_type	Gender
0	5.1	3.500000	1.400000	0.2	Iris-setosa	Male
1	4.9	3.000000	1.400000	0.2	Iris-setosa	Male
2	5.0	3.415217	1.463043	0.2	Iris-setosa	Male
3	5.4	3.415217	1.463043	0.4	Iris-setosa	Male
4	4.6	3.400000	1.400000	0.3	Iris-setosa	Male
...	...	...	...	...	...	...
143	6.7	3.000000	5.200000	2.3	Iris-virginica	Female
144	6.3	2.500000	5.000000	1.9	Iris-virginica	Female
145	6.5	3.000000	5.200000	2.0	Iris-virginica	Female
146	6.2	3.400000	5.400000	2.3	Iris-virginica	Female
147	5.9	3.000000	5.100000	1.8	Iris-virginica	Female

148 rows × 6 columns

In [ ]:

```
def func(s):
  if s == 'Male':
    return 0
  else:
    return 1

df['Sex'] = df.Gender.apply(func)
del df['Gender']
df
```

Out[ ]:

	sl	sw	pl	pw	flower_type	Sex
0	5.1	3.500000	1.400000	0.2	Iris-setosa	0
1	4.9	3.000000	1.400000	0.2	Iris-setosa	0
2	5.0	3.415217	1.463043	0.2	Iris-setosa	0
3	5.4	3.415217	1.463043	0.4	Iris-setosa	0
4	4.6	3.400000	1.400000	0.3	Iris-setosa	0
...	...	...	...	...	...	...
143	6.7	3.000000	5.200000	2.3	Iris-virginica	1
144	6.3	2.500000	5.000000	1.9	Iris-virginica	1
145	6.5	3.000000	5.200000	2.0	Iris-virginica	1
146	6.2	3.400000	5.400000	2.3	Iris-virginica	1
147	5.9	3.000000	5.100000	1.8	Iris-virginica	1

148 rows × 6 columns

Now, we may apply algorithms which take into consideration the 'Sex' column too.