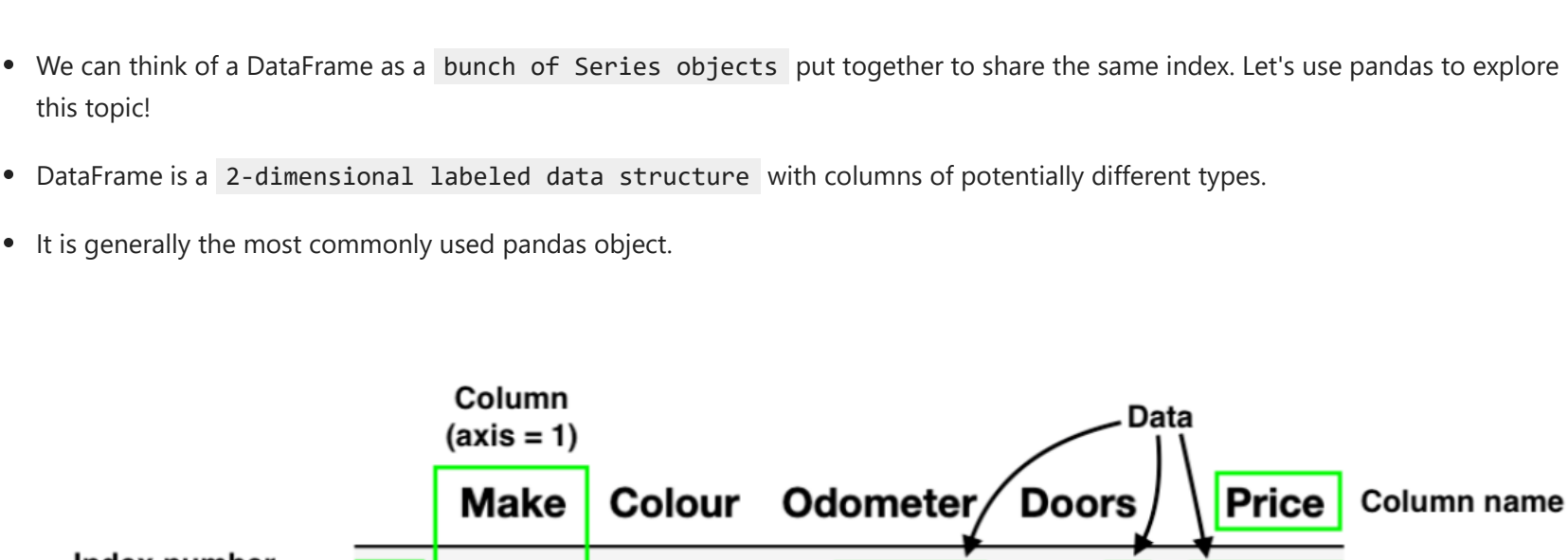
IOTA ACADEMY

Become Job Ready...

DataFrame

- We can think of a DataFrame as a `bunch of Series objects` put together to share the same index. Let's use pandas to explore this topic!
- DataFrame is a `2-dimensional labeled data structure` with columns of potentially different types.
- It is generally the most commonly used pandas object.



Importing the required libraries.

In [249]:
```
import pandas as pd
import numpy as np
```

1. Creating a DataFrame

`pd.DataFrame()`

You can create a dataframe by calling pd.DataFrame()

- **Syntax:** pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)
  - `data:` ndarray (structured or homogeneous), Iterable, dict, or DataFrame
  - `index:` . By default, if index is not passed and data provides no index, then integer indices will be used.
  - `columns:` - By default, if columns is not passed and data provides no column labels, then integer indices will be used.

1.1 Using 2-D numpy.ndarray

In [250]:
```
# create a 2-d array of ones
dt = np.ones((5,4))
df = pd.DataFrame(dt)
df
```

Out[250]:

	0	1	2	3
0	1.0	1.0	1.0	1.0
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0

In [251]:
```
# we can assign column or row labels

df= pd.DataFrame(dt,index=['a','b','c','d','e'],columns=["A","B","C","D"])
df
```

Out[251]:

	A	B	C	D
a	1.0	1.0	1.0	1.0
b	1.0	1.0	1.0	1.0
c	1.0	1.0	1.0	1.0
d	1.0	1.0	1.0	1.0
e	1.0	1.0	1.0	1.0

1.2 From dict of lists

In [252]:
```
dict1 = {"one":[1,2,3,4,5],"two":[6,7,8,9,10]}

df = pd.DataFrame(dict1)
df

# You can see the keys of the dictionary became the column headings (text in bold)
# no index is passed, the result will be range(n), where n is the array length.
```

Out[252]:

	one	two
0	1	6
1	2	7
2	3	8
3	4	9
4	5	10

In [253]:
```
# If an index is passed, it must also be of the same length as an array
df = pd.DataFrame(dict1,index=["a",'b','c','d','e'])
df
```

Out[253]:

	one	two
a	1	6
b	2	7
c	3	8
d	4	9
e	5	10

1.3 Using list of tuples

In [254]:
```
lt_of_tuple=[("BMW","Blue"),("Toyota","Red"),("Honda","White")]
df=pd.DataFrame(lt_of_tuple,columns=["cars","colour"])
df
```

Out[254]:

	cars	colour
0	BMW	Blue
1	Toyota	Red
2	Honda	White

The row and column labels can be accessed respectively by accessing the index and columns attributes:

- `dataframe_name.columns`
- `dataframe_name.index`

In [255]:
```
print(df.columns)

print(df.index)
```
```
Index(['cars', 'colour'], dtype='object')
RangeIndex(start=0, stop=3, step=1)
```

2. importing data

- Creating Series and DataFrame's from scratch is nice but what you'll usually be doing is importing your data in the form of a `.csv` (comma-separated value) or `spreadsheet file`.

2.1 From CSV file

`pd.read_csv()`

return a comma-separated values (csv) as two-dimensional data structure with labeled axes.

- **Syntax:** pd.read_csv('file_name.csv')

  `- The file name will be encoded as a string (in quotes)`

In [256]:
```
# Importing Employee salary dataset
# If you have a large DataFrame with many rows, Pandas will only return the first 5 rows and the last 5 rows:

df = pd.read_csv("Employee Salary Data11.csv")
df
```

Out[256]:

	Sr. No	Code	Employee Name	Designation	Department	Join Date	Monthly CTC	Annual CTC
0	1	BOM043	Employee_1	Junior Manager	Operations	02-09-2017	35030	420360
1	2	BOM063	Employee_3	Senior Executive	Operations	11-12-2017	34042	408504
2	3	BOM069	Employee_4	Senior Executive	Operations	01-01-2018	42150	505800
3	4	BOM056	Employee_10	Executive	Finance & Admin	13-03-2018	17856	214272
4	5	BOM145	Employee_13	Senior Executive	Operations	06-07-2018	48000	576000
...	...	...	...	...	...	...	...	...
101	102	BOM402	Employee_183	Executive	Marketing	17-05-2021	29950	359400
102	103	BOM403	Employee_185	Manager	Finance & Admin	17-05-2021	125000	1500000
103	104	BOM404	Employee_186	Senior Executive	Marketing	17-05-2021	40000	480000
104	105	BOM405	Employee_187	Senior Manager	Marketing	17-05-2021	159167	1910004
105	106	BOM406	Employee_188	Senior Executive	D2C	17-05-2021	28950	347400

106 rows × 8 columns

2.2 From Excel file

`pd.read_excel()`

Read an Excel file into a pandas DataFrame.

- **Syntax:** pd.read_excel('file_name.xlsx')

  `- The file name will be encoded as a string (in quotes)`

In [257]:
```
df = pd.read_excel("name_rollno.xlsx")
df
```

Out[257]:

	Name	Roll No
0	Abhishek	101
1	Shyam	102
2	Radha	103
3	Rani	104
4	Gautam	105
5	Kartik	106

In [258]:
```
# We can read individual sheets as well.
df = pd.read_excel("name_rollno.xlsx",sheet_name='Sheet2') # Use the sheet_name parameter to specify the name o
df
```

Out[258]:

	Name	Marks
0	Abhishek	50.0
1	Shyam	60.0
2	Radha	70.0
3	Rani	80.0
4	Gautam	90.0
5	Kartik	NaN

3. selection, addition, deletion of columns

3.1 column Selection

The easiest way to select a column of data is by using brackets `[ ]`

In [259]:
```
df = pd.DataFrame({"one":[1,2,3,4,5],
                   "two":[6,7,8,9,10],
                   "three":[11,12,13,12,17]},
                  index=['a','b','c','d','e'])
df
```

Out[259]:

	one	two	three
a	1	6	11
b	2	7	12
c	3	8	13
d	4	9	12
e	5	10	17

In [260]:
```
# Selection of a single column
# Put the column name inside a square bracket
df["one"]
```

Out[260]:
```
a    1
b    2
c    3
d    4
e    5
Name: one, dtype: int64
```

In [261]:
```
# Select multiple columns
# Pass a list of column names

df[["one","two"]]
```

Out[261]:

	one	two
a	1	6
b	2	7
c	3	8
d	4	9
e	5	10

In [262]:
```
# DOT notation method (Not Recommended)
# It will not work in many situations (for example- if there is space in column names)

df.one
```

Out[262]:
```
a    1
b    2
c    3
d    4
e    5
Name: one, dtype: int64
```

3.2 Create a new column:

In [263]:
```
df = pd.DataFrame({"one":[1,2,3,4,5],
                   "two":[6,7,8,9,10],
                   "three":[11,12,13,12,17]},
                  index=['a','b','c','d','e'])
df
```

Out[263]:

	one	two	three
a	1	6	11
b	2	7	12
c	3	8	13
d	4	9	12
e	5	10	17

In [264]:
```
# From existing columns
df['new'] = df['one'] + df['two']
```

In [265]:
```
df
```

Out[265]:

	one	two	three	new
a	1	6	11	7
b	2	7	12	9
c	3	8	13	11
d	4	9	12	13
e	5	10	17	15

In [266]:
```
# Example
df["four"] = df["one"] > 4

df
```

Out[266]:

	one	two	three	new	four
a	1	6	11	7	False
b	2	7	12	9	False
c	3	8	13	11	False
d	4	9	12	13	False
e	5	10	17	15	True

**When inserting a scalar value, it will naturally be propagated to fill the column:

In [267]:
```
# Example
df["five"] = 5
df
```

Out[267]:

	one	two	three	new	four	five
a	1	6	11	7	False	5
b	2	7	12	9	False	5
c	3	8	13	11	False	5
d	4	9	12	13	False	5
e	5	10	17	15	True	5

`insert()`

Insert column into DataFrame at the specified location.

Raises a ValueError if column is already contained in the DataFrame, unless allow_duplicates is set to True.

- **Syntax:** DataFrame.insert(loc, column, value, allow_duplicates=_NoDefault.no_default)

In [268]:
```
df
```

Out[268]:

	one	two	three	new	four	five
a	1	6	11	7	False	5
b	2	7	12	9	False	5
c	3	8	13	11	False	5
d	4	9	12	13	False	5
e	5	10	17	15	True	5

In [269]:
```
# df.insert(index, column_name, value)

df.insert(2,"location",[1,2,3,4,5])
```

In [270]:
```
df
```

Out[270]:

	one	two	location	three	new	four	five
a	1	6	1	11	7	False	5
b	2	7	2	12	9	False	5
c	3	8	3	13	11	False	5
d	4	9	4	12	13	False	5
e	5	10	5	17	15	True	5

3.3 Delete a column

In [271]:
```
df
```

Out[271]:

	one	two	location	three	new	four	five
a	1	6	1	11	7	False	5
b	2	7	2	12	9	False	5
c	3	8	3	13	11	False	5
d	4	9	4	12	13	False	5
e	5	10	5	17	15	True	5

In [272]:
```
del df["four"] # It will permanently delete the column.
```

In [273]:
```
df
```

Out[273]:

	one	two	location	three	new	five
a	1	6	1	11	7	5
b	2	7	2	12	9	5
c	3	8	3	13	11	5
d	4	9	4	12	13	5
e	5	10	5	17	15	5

Removing Columns

`drop()`

Remove rows or columns by specifying label names and corresponding axis, or by specifying direct index or column names.

- **Syntax:** DataFrame.drop(labels=None, *, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise'

In [274]:
```
df.drop(columns=['new']) # drop the column "new" from the DataFrame.
```

Out[274]:

	one	two	location	three	five
a	1	6	1	11	5
b	2	7	2	12	5
c	3	8	3	13	5
d	4	9	4	12	5
e	5	10	5	17	5

In [275]:
```
df  # Notice that the column has not been deleted from the original DataFrame.
```

Out[275]:

	one	two	location	three	new	five
a	1	6	1	11	7	5
b	2	7	2	12	9	5
c	3	8	3	13	11	5
d	4	9	4	12	13	5
e	5	10	5	17	15	5

In [276]:
```
# To delete a column from the original DataFrame, we must set the inplace argument to true.
df.drop(columns=['new'],inplace=True)
```

In [277]:
```
df  # The column 'new' has been removed from the original dataframe.
```

Out[277]:

	one	two	location	three	five
a	1	6	1	11	5
b	2	7	2	12	5
c	3	8	3	13	5
d	4	9	4	12	5
e	5	10	5	17	5

4. selection, addition, deletion of rows

4.1 Selecting rows

- Access a group of rows and columns by label(s), index, or a Boolean array.

`a) loc[]`

loc is used to select rows and columns by Names/Labels

- **Syntax:** DataFrame.loc[]

In [278]:
```
# Creating a dataframe
values = [['Rohan',455],['Elvish',250],['Deepak',495],
          ['Soni',400],['Radhika',350],['Vansh',450]]

df1 = pd.DataFrame(values,columns=["Name","Number"],index=list("abcdef"))
df1
```

Out[278]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

In [279]:
```
df1.loc['a']  # row label
```

Out[279]:
```
Name      Rohan
Number      455
Name: a, dtype: object
```

In [280]:
```
# Pass a list of row labels
df1.loc[['a','c']]
```

Out[280]:

	Name	Number
a	Rohan	455
c	Deepak	495

Select row and column both together

In [281]:
```
df1.loc["a","Name"]
```

Out[281]:
```
'Rohan'
```

You can also do slicing

- Note: start and stop of the slice are included.

In [282]:
```
df1.loc["a":"e"] # both are included
```

Out[282]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350

In [283]:
```
df1.loc["b":] # label 'b' to the end
```

Out[283]:

	Name	Number
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

In [284]:
```
df1.loc['b':"d","Name"] # label 'b' to 'd' and single column 'Name'
```

Out[284]:
```
b    Elvish
c    Deepak
d      Soni
Name: Name, dtype: object
```

In [285]:
```
df1.loc["b":,:]  # label 'b' to the end and all columns
```

Out[285]:

	Name	Number
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

In [286]:
```
df1.loc["a":"d","Number"]
```

Out[286]:
```
a    455
b    250
c    495
d    400
Name: Number, dtype: int64
```

In [287]:
```
# Boolean list with the same length as the row axis
# It will return only those rows where True

df1.loc[[True,False,True,False,True,False]]
```

Out[287]:

	Name	Number
a	Rohan	455
c	Deepak	495
e	Radhika	350

`b) iloc[]`

iloc is a method that is used to select rows and columns by position/index. If the position/index does not exist, it gives an index error

- **Syntax:** DataFrame.iloc[]

In [288]:
```
# Creating a dataframe
values = [['Rohan',455],['Elvish',250],['Deepak',495],
          ['Soni',400],['Radhika',350],['Vansh',450]]

df1 = pd.DataFrame(values,columns=["Name","Number"],index=list("abcdef"))
df1
```

Out[288]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

In [289]:
```
df1.iloc[0]    # first row
```

Out[289]:
```
Name      Rohan
Number      455
Name: a, dtype: object
```

In [290]:
```
df1.iloc[[0,2]]  # pass multiple indices in a list
```

Out[290]:

	Name	Number
a	Rohan	455
c	Deepak	495

Select row and column both together

In [291]:
```
df1.iloc[0,0]
```

Out[291]:
```
'Rohan'
```

You can also do slicing

- Note: start(included) and stop(excluded)

In [292]:
```
df1.iloc[:4]   # from 0 to 3
```

Out[292]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400

Use `:` to select the entire axis.

In [293]:
```
df1.iloc[:4,:]  # row indices 0 to 3 and all the columns
```

Out[293]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400

In [294]:
```
df1.iloc[:,0]   #  all rows and first column
```

Out[294]:
```
a      Rohan
b     Elvish
c     Deepak
d       Soni
e    Radhika
f      Vansh
Name: Name, dtype: object
```

In [295]:
```
# using a boolean mask of the index's length.

df1.iloc[[True,False,True,True,False,True]]  # it will give only those rows where boolean value is true
```

Out[295]:

	Name	Number
a	Rohan	455
c	Deepak	495
d	Soni	400
f	Vansh	450

4.2 deleting rows

In [296]:
```
df1
```

Out[296]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

`DataFrame.drop()`

- Drop specified labels from rows or columns.

In [297]:
```
df1.drop("a",axis=0)  # axis-->0 rows
                      # axis--->1 colums
```

Out[297]:

	Name	Number
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

In [298]:
```
df1  # Notice that the row has not been deleted from the original DataFrame.
```

Out[298]:

	Name	Number
a	Rohan	455
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

In [299]:
```
# To delete a column from the original DataFrame, we must set the inplace argument to true.
df1.drop("a",axis=0,inplace=True)
```

In [300]:
```
df1
```

Out[300]:

	Name	Number
b	Elvish	250
c	Deepak	495
d	Soni	400
e	Radhika	350
f	Vansh	450

5. Conditional Selection

An important feature of pandas is conditional selection using bracket notation, very similar to NumPy:

In [301]:
```
df = pd.DataFrame({"Name":["Rahul","Shyam","Lalit","Ajay","Harish","Sonu"],
             "TotalMarks":[82, 80, 63, 22, 65, 40],
             "Grade": ['A', 'E', 'B','E','C','D'],
            "Subjects":["Math","Biology","Math","Maths","Commerce","Arts"]})
```

In [302]:
```
df
```

Out[302]:

	Name	TotalMarks	Grade	Subjects
0	Rahul	82	A	Math
1	Shyam	80	E	Biology
2	Lalit	63	B	Math
3	Ajay	22	E	Maths
4	Harish	65	C	Commerce
5	Sonu	40	D	Arts

Let's extract data based on the condition of when the marks are more than 45 in the TotalMarks column.

In [303]:
```
df['TotalMarks']>45

# The output gives boolean values instead of records. It means where it finds condition is matched,
# it returns True, and if not, it returns False.
```

Out[303]:
```
0     True
1     True
2     True
3    False
4     True
5    False
Name: TotalMarks, dtype: bool
```

In order to get rows with data that meets our condition, we have to **apply the condition in a data frame**, and this will give us our expected result.

In [304…
```
df[df["TotalMarks"]>45]
```

Out[304]:

	Name	TotalMarks	Grade	Subjects
0	Rahul	82	A	Math
1	Shyam	80	E	Biology
2	Lalit	63	B	Math
4	Harish	65	C	Commerce

In [305…
```
# Example
df[df["Subjects"]=="Math"]
```

Out[305]:

	Name	TotalMarks	Grade	Subjects
0	Rahul	82	A	Math
2	Lalit	63	B	Math

In [306…
```
# after applying the condition, choose the required columns.

df[df["TotalMarks"]>45][["Name","Subjects"]] # Pass multiple column names in a list
```

Out[306]:

	Name	Subjects
0	Rahul	Math
1	Shyam	Biology
2	Lalit	Math
4	Harish	Commerce

using multiple conditions

- For two or more conditions you can use **|** (for **or** operator) and **&** (for **and** operator) with parenthesis:

In [307…
```
df[(df["TotalMarks"]>80)&(df["Subjects"]=="Math")]

# Put both conditions within parentheses.
```

Out[307]:

	Name	TotalMarks	Grade	Subjects
0	Rahul	82	A	Math

In [308…
```
# Example
df[(df["Subjects"]=="Math") | (df["TotalMarks"]>60)]
```

Out[308]:

	Name	TotalMarks	Grade	Subjects
0	Rahul	82	A	Math
1	Shyam	80	E	Biology
2	Lalit	63	B	Math
4	Harish	65	C	Commerce

6) Arithmetic operations

Arithmetic operations with scalars operate element-wise:

In [309…
```
df = pd.DataFrame({"W":[4,4,4],
                   "X":[0,2,4],
                   "Y":[3,3,4],
                   "Z":[1,0,2]},index=["A","B","C"])
df
```

Out[309]:

	W	X	Y	Z
A	4	0	3	1
B	4	2	3	0
C	4	4	4	2

In [310…
```
df*5   # All values will be multiplied by 5
```

Out[310]:

	W	X	Y	Z
A	20	0	15	5
B	20	10	15	0
C	20	20	20	10

In [311…
```
df/2
```

Out[311]:

	W	X	Y	Z
A	2.0	0.0	1.5	0.5
B	2.0	1.0	1.5	0.0
C	2.0	2.0	2.0	1.0

In [312…
```
df**2
```

Out[312]:

	W	X	Y	Z
A	16	0	9	1
B	16	4	9	0
C	16	16	16	4

In [313…
```
# with single column
df["W"]*2
```

Out[313]:
```
A    8
B    8
C    8
Name: W, dtype: int64
```

7. Transposing

To transpose, access the `T` attribute or `DataFrame.transpose()`, similar to an ndarray:

In [314…
```
df.T  # Transpose index and columns.
```

Out[314]:

	A	B	C
W	4	4	4
X	0	2	4
Y	3	3	4
Z	1	0	2

8. More Index Details

Let's discuss some more features of indexing, including resetting the index or setting it to something else. We'll also talk about index hierarchy!

In [315…
```
df = pd.DataFrame({'month': [1, 4, 7, 10],
                   'year': [2012, 2014, 2013, 2014],
                   'sale': [55, 40, 84, 31]})
```

In [316…
```
df
```

Out[316]:

	month	year	sale
0	1	2012	55
1	4	2014	40
2	7	2013	84
3	10	2014	31

`set_index()`

The set_index() method allows one or more column values to become the row index.

- **Syntax:** dataframe.set_index(keys, drop, append, inplace, verify_integrity)

In [317…
```
# A certain column can be set as the index.
df.set_index("year")

# the changes will not happen in the original DataFrame, we must set the inplace argument to true.
```

Out[317]:

	month	sale
year		
2012	1	55
2014	4	40
2013	7	84
2014	10	31

`reset_index()`

The reset_index() method allows you to reset the index back to the default 0, 1, 2, etc. indexes.

- **Syntax:** dataframe.reset_index(level, drop, inplace, col_level, col_fill)

In [318…
```
df = pd.DataFrame([('bird', 389.0),
                   ('bird', 24.0),
                   ('mammal', 80.5),
                   ('mammal', np.nan)],
                  index=['falcon', 'parrot', 'lion', 'monkey'],
                  columns=('class', 'max_speed'))

df
```

Out[318]:

	class	max_speed
falcon	bird	389.0
parrot	bird	24.0
lion	mammal	80.5
monkey	mammal	NaN

Note:-When we reset the index, the old index is added as a column, and a new sequential index is used:

In [319…
```
# Reset to default 0,1...n index
df.reset_index()
```

Out[319]:

	index	class	max_speed
0	falcon	bird	389.0
1	parrot	bird	24.0
2	lion	mammal	80.5
3	monkey	mammal	NaN

In [320…
```
# We can use the drop parameter to avoid the old index being added as a column:
df.reset_index(drop=True)
```

Out[320]:

	class	max_speed
0	bird	389.0
1	bird	24.0
2	mammal	80.5
3	mammal	NaN

9. Multi-Index and Index Hierarchy

Let us go over how to work with Multi-Index, first we'll create a quick example of what a Multi-Indexed DataFrame would look like:

In [321…
```
# Index Levels
outside = ['G1','G1','G1','G2','G2','G2']
inside = [1,2,3,1,2,3]
hier_index = list(zip(outside,inside))

print(hier_index)
hier_index = pd.MultiIndex.from_tuples(hier_index)
type(hier_index)
```

```
[('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2), ('G2', 3)]
```

Out[321]:
```
pandas.core.indexes.multi.MultiIndex
```

In [322…
```
hier_index
```

Out[322]:
```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           )
```

In [323…
```
df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
```

Out[323]:

		A	B
G1	**1**	-0.479519	-0.489874
	2	0.104019	-0.894685
	3	0.121749	0.988312
G2	**1**	0.149728	-0.068850
	2	-0.176049	0.282254
	3	-1.213611	-0.442633

Now let's show how to index this! For index hierarchy we use df.loc[], if this was on the columns axis, you would just use normal bracket notation df[]. Calling one level of the index returns the sub-dataframe:

In [324…
```
df.loc['G1']
```

Out[324]:

	A	B
1	-0.479519	-0.489874
2	0.104019	-0.894685
3	0.121749	0.988312

In [325…
```
df.loc['G1'].loc[1]
```

Out[325]:
```
A   -0.479519
B   -0.489874
Name: 1, dtype: float64
```

In [326…
```
df.index.names
```

Out[326]:
```
FrozenList([None, None])
```

In [327…
```
df.index.names = ['Group','Num']
```

In [328…
```
df
```

Out[328]:

		A	B
Group	**Num**		
G1	**1**	-0.479519	-0.489874
	2	0.104019	-0.894685
	3	0.121749	0.988312
G2	**1**	0.149728	-0.068850
	2	-0.176049	0.282254
	3	-1.213611	-0.442633

In [329…
```
df.xs('G1')
```

Out[329]:

	A	B
Num		
1	-0.479519	-0.489874
2	0.104019	-0.894685
3	0.121749	0.988312

In [330…
```
df.xs(['G1',1])
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_8\580597333.py:1: FutureWarning: Passing lists as key for xs is deprecated and will be removed in a future version. Pass key as a tuple instead.
  df.xs(['G1',1])
```

Out[330]:
```
A   -0.479519
B   -0.489874
Name: (G1, 1), dtype: float64
```

In [331…
```
df.xs(1,level='Num')
```

Out[331]:

	A	B
Group		
G1	-0.479519	-0.489874
G2	0.149728	-0.068850

Great Job!