PROFESSIONAL ELECTIVE COURSES: VERTICALS
VERTICAL 1: DATA SCIENCE
CCS346 EXPLORATORY DATA ANALYSIS

UNIT II - EDA USING PYTHON

Data Manipulation With Pandas – Pandas Objects - Data Indexing And Selection – Operating On Data – Handling Missing Data – Hierarchical Indexing – Combining Datasets –Concat, Append, Merge And Join - Aggregation And Grouping – Pivot Tables – Vectorized String Operations.

What is Pandas?

Pandas is a Python library used for working with data sets. It has functions for analysing, cleaning, exploring, and manipulating data. The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyse big data and make conclusions based on statistical theories. Pandas can clean messy data sets, and make them readable and relevant. Relevant data is very important in data science.

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data. In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.

Prior to Pandas, Python was majorly used for data mining and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data -load, prepare, manipulate, model, and analyse.

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Key Features of Pandas

- Fast and efficient Data Frame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and sub-setting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Standard Python distribution doesn't come bundled with Pandas module. A lightweight alternative is to install NumPy using popular Python package installer, pip.

pip install pandas

If you install Anaconda Python package, Pandas will be installed by default with the following –

Windows

- **Anaconda** (from https://www.continuum.io) is a free Python distribution for SciPy stack. It is also available for Linux and Mac.
- **Canopy** (https://www.enthought.com/products/canopy/) is available as free as well as commercial distribution with full SciPy stack for Windows, Linux and Mac.
- **Python** (x,y) is a free Python distribution with SciPy stack and Spyder IDE for Windows OS. (Downloadable from http://python-xy.github.io/)

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Series

- Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

10	23	56	17	52	61	73	90	26	72

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

DataFrame

- DataFrame is a two-dimensional array with heterogeneous data. For example, the table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

Data Type of Columns

The data types of the four columns are as follows –

Column	Type
Name	String
Age	Integer
Gender	String
Rating	Float

Panel

- Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

WORKING WITH SERIES

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

Creating Series

1. Create an Empty Series

- A basic series, which can be created is an Empty Series.

#import the pandas library and aliasing as pd import pandas as pd s = pd.Series() print (s)	Series([], dtype: float64)

2. Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., [0,1,2,3…. **range(len(array))-1].**

import pandas as pd import numpy as np data = np.array(['a','b','c','d']) s = pd.Series(data) print (s)	0 a 1 b 2 c 3 d dtype: object
import pandas as pd import numpy as np data = np.array(['a','b','c','d']) s = pd.Series(data,index=[100,101,102,103]) print (s)	100 a 101 b 102 c 103 d dtype: object

3. Create a Series from dict

A dict can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If index is passed, the values in data corresponding to the labels in the index will be pulled out.

import pandas as pd import numpy as np data = {'a' : 0., 'b' : 1., 'c' : 2.} s = pd.Series(data) print (s)	a 0.0 b 1.0 c 2.0 dtype: float64
import pandas as pd import numpy as np data = {'a' : 0., 'b' : 1., 'c' : 2.} s = pd.Series(data,index=['b','c','d','a']) print (s)	b 1.0 c 2.0 d NaN a 0.0 dtype: float64

4. Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of index

import pandas as pd	0 5

import numpy as np s = pd.Series(5, index=[0, 1, 2, 3]) print (s)	1 5 2 5 3 5 dtype: int64
Accessing Data from Series with Position	
1. ***Data in the series can be accessed similar to that in an ndarray.***	
import pandas as pd s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e']) #retrieve the first element print (s[0])	1
2. Retrieve the first three elements in the Series	
import pandas as pd s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e']) #retrieve the first three elements print (s[0:3])	a 1 b 2 c 3 dtype: int64
3. ***Retrieve the last three elements.***	
import pandas as pd s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e']) #retrieve the last three element print (s[-3:])	c 3 d 4 e 5 dtype: int64
4. ***Retrieve multiple elements using a list of index label values.***	
import pandas as pd s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e']) #retrieve multiple elements print s[['a','c','d']]	a 1 c 3 d 4 dtype: int64

WORKING WITH DATA FRAMES

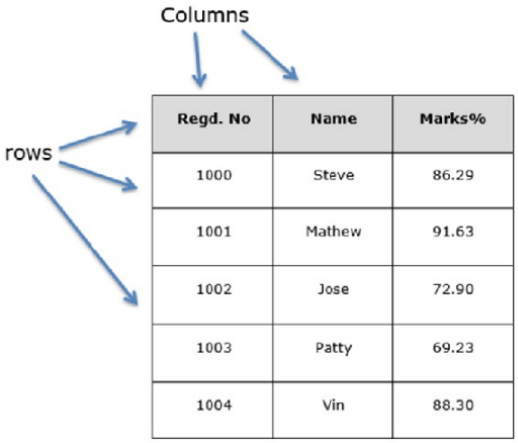
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Structure

- Let us assume that we are creating a data frame with student's data.



pandas.DataFrame

- A pandas DataFrame can be created using the following constructor –

pandas.DataFrame(data, index, columns, dtype, copy)

Create DataFrame

- A pandas DataFrame can be created using various inputs like –
  - Lists
  - Dict
  - Series
  - Numpy ndarrays
  - Another DataFrame

1. ***Create an Empty DataFrame***	
#import the pandas library and aliasing as pd import pandas as pd df = pd.DataFrame() print (df)	Empty DataFrame Columns: [] Index: []
2. Create a DataFrame from Lists The DataFrame can be created using a single list or a list of lists.	
import pandas as pd data = [1,2,3,4,5] df = pd.DataFrame(data) print (df)	0 0 1 1 2 2 3 3 4 4 5
import pandas as pd data = [['Alex',10],['Bob',12],['Clarke',13]] df = pd.DataFrame(data,columns=['Name','Age']) print (df)	Name Age 0 Alex 10 1 Bob 12 2 Clarke 13
import pandas as pd data = [['Alex',10],['Bob',12],['Clarke',13]] df = pd.DataFrame(data,columns=['Name','Age'],dtype=float) print df	Name Age 0 Alex 10.0 1 Bob 12.0 2 Clarke 13.0

3. Create a DataFrame from Dict of ndarrays / Lists All the ndarrays must be of same length. If index is passed, then the length of the index should equal to the length of the arrays. If no index is passed, then by default, index will be range(n), where n is the array length.	
import pandas as pd data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'], 'Age':[28,34,29,42]} df = pd.DataFrame(data) print (df)	Name Age 0 Tom 28 1 Jack 34 2 Steve 29 3 Ricky 42
4. ***Create an indexed DataFrame using arrays.***	
import pandas as pd data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]} df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4']) print (df)	Name Age rank1 Tom 28 rank2 Jack 34 rank3 Steve 29 rank4 Ricky 42
5. Create a DataFrame from List of Dicts	
import pandas as pd data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}] df = pd.DataFrame(data) print (df)	a b c 0 1 2 NaN 1 5 10 20.0

import pandas as pd data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}] df = pd.DataFrame(data, index=['first', 'second']) print (df)	a b c first 1 2 NaN second 5 10 20.0
import pandas as pd data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}] *#With two column indices, values same as dictionary keys* df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b']) *#With two column indices with one index with other name* df2 = pd.DataFrame(data, index=['first', 'second'], columns=['x', 'y']) print (df1) print (df2)	a b first 1 2 second 5 10 x y first NaN NaN second NaN NaN
Column Selection We will understand this by selecting a column from the DataFrame.	
import pandas as pd d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']), 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])} df = pd.DataFrame(d) print (df ['one'])	a 1.0 b 2.0 c 3.0 d NaN Name: one, dtype: float64
6. Column Addition is performed by adding a new column to an existing data frame.	

```
import pandas as pd

d = {'one' : pd.Series([11, 22, 33, 44], index=['a', 'b', 'c', 'd']),
 'two' : pd.Series([100, 200, 300,400], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print ("Given Data Frame:")
print(df)

# Adding a new column to an existing DataFrame object with column label by passing new series

print ("Adding a new column by passing as Series:")
df['three']=pd.Series([7,8,9],index=['a','b','c'])
print (df)

print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']

print (df)
```

```
Given Data Frame:
   one  two
a   11  100
b   22  200
c   33  300
d   44  400

Adding a new column by passing as Series:
   one  two  three
a   11  100    7.0
b   22  200    8.0
c   33  300    9.0
d   44  400    NaN

Adding a new column using the existing columns in DataFrame:
   one  two  three  four
a   11  100    7.0  18.0
b   22  200    8.0  30.0
c   33  300    9.0  42.0
d   44  400    NaN   NaN
```

7. Columns can be deleted or popped

```
import pandas as pd

d = {'one' : pd.Series([11, 22, 33], index=['a', 'b', 'c']),
  'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
  'three' : pd.Series([10,20,30], index=['a','b','c'])}
df = pd.DataFrame(d)
print ("Our dataframe is:")
print (df)

# using del function
print ("Deleting using DEL function:")
del df['two']
print (df)

# using pop function
print ("Deleting using POP function:")
df.pop('one')
print (df)
```

```
Our dataframe is:
   one  two  three
a   11    1     10
b   22    2     20
c   33    3     30

Deleting using DEL function:
   one  three
a   11     10
b   22     20
c   33     30

Deleting using POP function:
   three
a     10
b     20
c     30
```

Row Selection, Addition, and Deletion

8. Rows can be selected by passing row label to a loc function.

```
import pandas as pd
d = {'one' : pd.Series([11, 22, 33], index=['a', 'b', 'c']),
  'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}

df = pd.DataFrame(d)
print (df)
```

```
    one  two
a    11    1
b    22    2
c    33    3

one    22
```

print (**df.loc['b']**)	two 2 Name: b, dtype: int64
import pandas as pd d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']), 'two' : pd.Series([100, 200, 300], index=['a', 'b', 'c'])} df = pd.DataFrame(d) print (df.iloc[2])	one 3 two 300 Name: c, dtype: int64
9. Slice Rows Multiple rows can be selected using ' : ' operator.	
import pandas as pd d = {'one' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']), 'two' : pd.Series([100, 200, 300, 400], index=['a', 'b', 'c', 'd'])} df = pd.DataFrame(d) print (df[2:4])	one two c 3 300 d 4 400
10. Addition of Rows Add new rows to a DataFrame using the append function.	
import pandas as pd df = pd.DataFrame([[55, 66], [77, 66]], columns = ['a','b']) df2 = pd.DataFrame([[700, 600], [800, 900]], columns = ['a','b']) df = df.append(df2) print (df)	a b 0 55 66 1 77 66 0 700 600 1 800 900
11. Deletion of Rows Drop a label and see how many rows will get dropped.	
import pandas as pd df = pd.DataFrame([[11, 22], [33, 44]], columns = ['a','b']) df2 = pd.DataFrame([[55, 66], [77, 88]], columns = ['a','b']) df = df.append(df2) print("Original Data frame.....") print(df) print("Drop rows with label 0.....") df = df.drop(0) print (df)	Original Data frame..... a b 0 11 22 1 33 44 0 55 66 1 77 88 Drop rows with label 0..... a b 1 33 44 1 77 88
Descriptive Statistics	

import pandas as pd
import numpy as np
 	
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
'Lee','David','Gasper','Betina','Andres']),
'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
 	
'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}
	Name Age Rating
	0 Tom 25 4.23
	1 James 26 3.24
	2 Ricky 25 3.98
	3 Vin 23 2.56
	4 Steve 30 3.20
	5 Smith 29 4.60
	6 Jack 23 3.80
	7 Lee 34 3.78
	8 David 40 2.98
	9 Gasper 30 4.80

#Create a DataFrame
df = pd.DataFrame(d)
print (df)
	10 Betina 51 4.10
	11 Andres 46 3.65

sum() Returns the sum of the values for the requested axis. By default, axis is index **(axis=0).**	

import pandas as pd
import numpy as np
 	
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
'Lee','David','Gasper','Betina','Andres']),
'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}
 	
#Create a DataFrame
df = pd.DataFrame(d)
print (df.sum())
	Name
	TomJamesRickyVinSteveSmithJackLeeDavidGasperBe...
	Age 382
	Rating 44.92
	dtype: object
	
	Note:
	Each individual column is added individually (Strings are appended).

axis=1	

import pandas as pd
import numpy as np
 	
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
'Lee','David','Gasper','Betina','Andres']),
'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}
 	
#Create a DataFrame
df = pd.DataFrame(d)
print (df.sum(1))
	0 29.23
	1 29.24
	2 28.98
	3 25.56
	4 33.20
	5 33.60
	6 26.80
	7 37.78
	8 42.98
	9 34.80
	10 55.10
	11 49.65
	dtype: float64

import pandas as pd
import numpy as np
 	
#Create a Dictionary of series
d =
{'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46])}
 	
#Create a DataFrame
df = pd.DataFrame(d)
print("Mean .....")
print (df.mean())
	Mean .....
	Age 31.833333
	dtype: float64

import pandas as pd
import numpy as np
 	
#Create a Dictionary of series
d =
{'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46])}
 	
#Create a DataFrame
df = pd.DataFrame(d)
print("standard Devaiation .....")
print (df.std())
	Standard Devaiation .....
	Age 9.232682
	dtype: float64

```
import pandas as pd
import numpy as np

#Create a Dictionary of series
d =
{'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46])}

#Create a DataFrame
df = pd.DataFrame(d)
print("Describing various datas.....")
print (df.describe())
```

```
Describing various datas.....
Age
count  12.000000
mean   31.833333
std    9.232682
min    23.000000
25%    25.000000
50%    29.500000
75%    35.500000
max    51.000000
```

Drop duplicates

```
# import the library as pd
import pandas as pd
df = pd.DataFrame(
   {
      'Name': ['Srivignesh', 'Srivignesh','Hari'],
      'Age': [22,22, 11],
      'Country': ['India','India', 'India']
   }
)
print(df)
newdf = df.drop_duplicates()
print(newdf)
```

```
   Name       Age Country
0  Srivignesh   22   India
1  Srivignesh   22   India
2     Hari      11   India

      Name       Age Country
0     Srivignesh   22   India
      2          Hari         11   India
```

— **Transposed Data Frame**
  - The transpose() function is used to transpose index and columns

— **axes**
  - Returns the list of row axis labels and column axis labels.

— **empty**
  - Returns the Boolean value saying whether the Object is empty or not; True indicates that the object is empty.

— **ndim**
  - Returns the number of dimensions of the object. By definition, DataFrame is a 2D object.

— **Shape**
  - Returns a tuple representing the dimensionality of the DataFrame. Tuple (a,b), where a represents the number of rows and b represents the number of columns.

— **Size**
  - Returns the number of elements in the DataFrame.

— **values**
  - Returns the actual data in the DataFrame as an NDarray

— **Head & Tail**
  - To view a small sample of a DataFrame object, use the head() and tail() methods.
  - ***head()*** returns the first n rows (observe the index values). The default number of elements to display is five, but you may pass a custom number.

- ***tail()*** returns the last n rows (observe the index values). The default number of elements to display is five, but you may pass a custom number.

```
import pandas as pd
import numpy as np

#Create a Dictionary
df = pd.DataFrame(
  {'Name':['Tom','James','Ricky','Vin','Steve','Smith','Jack'],
  'Age':[25,26,25,23,30,29,23],
  'Rating':[4.23,3.24,3.98,2.56,3.20,4.6,3.8]})

#print the DataFrame

print ("Our Data Frame:")
print (df)

print ("Transposed Data Frame:")
print (df.T)

print ("Row axis labels and column axis labels are:")
print (df.axes)

print ("The data types of each column are:")
print (df.dtypes)

print ("Is the object empty?")
print (df.empty)

print ("The dimension of the object is:")
print (df.ndim)

print ("The shape of the object is:")
print (df.shape)

print ("The total number of elements in our object is:")
print (df.size)

print ("The actual data in our data frame is:")
print (df.values)

print ("The first two rows of the data frame is:")
print (df.head(2))
```

Our Data Frame:

	Name	Age	Rating
0	Tom	25	4.23
1	James	26	3.24
2	Ricky	25	3.98
3	Vin	23	2.56
4	Steve	30	3.20
5	Smith	29	4.60
6	Jack	23	3.80

Transposed Data Frame:

	0	1	2	3	4	5	6
Name	Tom	James	Ricky	Vin	Steve	Smith	Jack
Age	25	26	25	23	30	29	23
Rating	4.23	3.24	3.98	2.56	3.2	4.6	3.8

Row axis labels and column axis labels are:

[RangeIndex(start=0, stop=7, step=1), Index(['Name', 'Age', 'Rating'], dtype='object')]

The data types of each column are:

```
Name      object
Age        int64
Rating   float64
dtype: object
```

Is the object empty?
False

The dimension of the object is:
2
The shape of the object is:
(7, 3)

The total number of elements in our object is:
21

The actual data in our data frame is:

```
[['Tom' 25 4.23]
 ['James' 26 3.24]
 ['Ricky' 25 3.98]
 ['Vin' 23 2.56]
 ['Steve' 30 3.2]
 ['Smith' 29 4.6]
 ['Jack' 23 3.8]]
```

The first two rows of the data frame is:

	Name	Age	Rating
0	Tom	25	4.23
1	James	26	3.24

The first five rows of the data frame is:

# if value not specified, default is taken as 5 print ("The first five rows of the data frame is:") print (df.head()) **print ("The last two rows of the data frame is:")** print (df.tail(2))	Name Age Rating 0 Tom 25 4.23 1 James 26 3.24 2 Ricky 25 3.98 3 Vin 23 2.56 4 Steve 30 3.20 The last two rows of the data frame is: Name Age Rating 5 Smith 29 4.6 6 Jack 23 3.8

Read CSV Files

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.
- In our examples we will be using a CSV file called 'data.csv'.

/* Load the CSV into a DataFrame */

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.to_string())
```

Tip: use to_string() to print the entire DataFrame.

/*Print Data Frame without the `to_string()` method */

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
..	...	...	...	...
164	60	105	140	290.8
165	60	110	145	300.4
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

[169 rows x 4 columns]

max_rows

The number of rows returned is defined in Pandas option settings. We can check your system's maximum rows with the **`pd.options.display.max_rows`** statement.

/* Python program to cheack maximum rows */

```
import pandas as pd
print(pd.options.display.max_rows)
```

60

In my system the number is 60, which means that if the DataFrame contains more than 60 rows, the print(df) statement will return only the headers and the first and last 5 rows.

You can change the maximum rows number with the same statement.

/* Increase the maximum number of rows to display the entire DataFrame */

```
import pandas as pd
pd.options.display.max_rows = 9999
df = pd.read_csv('data.csv')
```

APEC

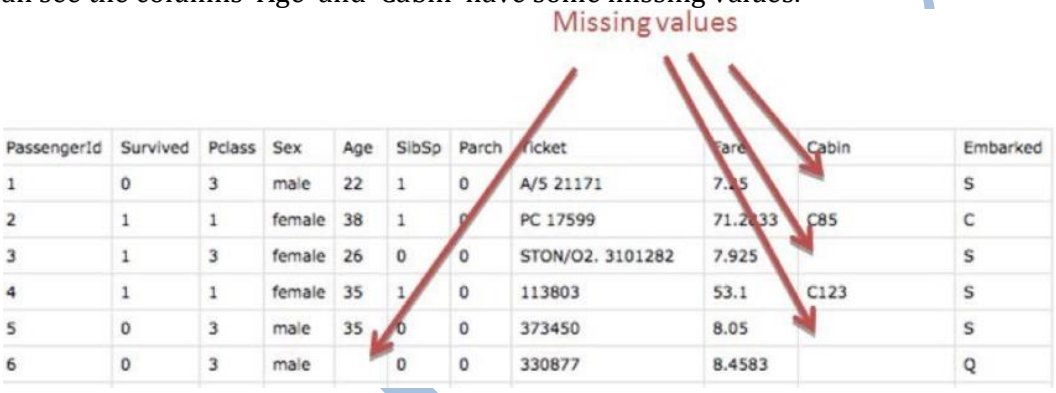
```
print(df)
```

Missing Data in Pandas

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. Missing Data can occur when no information is provided for one or more items or for a whole unit. Missing Data is a very big problem in real-life scenarios.

What is a Missing Value?

Missing data is defined as the values or data that is not stored (or not present) for some variable/s in the given dataset. Below is a sample of the missing data from the Titanic dataset. You can see the columns 'Age' and 'Cabin' have some missing values.



PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	male	22	1	0	A/5 21171	7.25		S
2	1	1	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	female	35	1	0	113803	53.1	C123	S
5	0	3	male	35	0	0	373450	8.05		S
6	0	3	male		0	0	330877	8.4583		Q

In DataFrame sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. To make matters even more complicated, different data sources may indicate missing data in different ways. For example, suppose different users being surveyed may choose not to share their income, some users may choose not to share the address in this way many datasets went missing.

Trade-Offs in Missing Data Conventions

A number of schemes have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a mask that globally indicates missing values, or choosing a sentinel value that indicates a missing entry.

- In the masking approach : The mask might be an Boolean array
- In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number)

Pandas treat None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame :

- isnull()
- notnull()
- dropna()

- fillna()
- replace()
- interpolate()

Checking for missing values using isnull() and notnull()

In order to check missing values in Pandas DataFrame, we use a function isnull() and notnull(). Both function help in checking whether a value is NaN or not. These functions can also be used in Pandas Series in order to find null values in a series.

/* Python program to check missing values */

```
import pandas as pd
import numpy as np

# dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from list
df = pd.DataFrame(dict)
print (df)

# using isnull() function
df.isnull()
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	NaN	56.0	80.0
3	95.0	NaN	98.0

	First Score	Second Score	Third Score
0	False	False	True
1	False	False	False
2	True	False	False
3	False	True	False

When `df.notnull()` is applied

	First Score	Second Score	Third Score
0	True	True	False
1	True	True	True
2	False	True	True
3	True	False	True

Dropping missing values using dropna()

In order to drop a null values from a dataframe, we used dropna() function this function drop Rows/Columns of datasets with Null values in different ways.

Code #1: Dropping rows with at least 1 null value.

```
# importing pandas as pd
import pandas as pd

# importing numpy as np
import numpy as np

# dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
        'Second Score': [30, np.nan, 45, 56],
        'Third Score':[52, 40, 80, 98],
        'Fourth Score':[np.nan, np.nan, np.nan,65]}

# creating a dataframe from dictionary
df = pd.DataFrame(dict)
```

Output

Data Frame:

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52	NaN
1	90.0	NaN	40	NaN
2	NaN	45.0	80	NaN
3	95.0	56.0	98	65.0

After dropping rows with atleast 1 NaN value

	First Score	Second Score	Third Score	Fourth Scor
3	95.0	56.0	98	65.

```
# using dropna() function
df.dropna()
```

Code #2: Drop rows whose all data is missing or contain null values(NaN)

```
# importing pandas as pd
import pandas as pd

# importing numpy as np
import numpy as np

# dictionary of lists
dict = {'First Score':[100, np.nan, np.nan, 95],
        'Second Score': [30, np.nan, 45, 56],
        'Third Score':[52, np.nan, 80, 98],
        'Fourth Score':[np.nan, np.nan, np.nan, 65]}

df = pd.DataFrame(dict)
print(df)
# using dropna() function
df.dropna(how = 'all')
```

Output

Data Frame:

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
1	NaN	NaN	NaN	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

After dropping rows whose all data is missing or contain null values(NaN)

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

Filling missing values using fillna(), replace() and interpolate()

In order to fill null values in a datasets, we use fillna(), replace() and interpolate() function these function replace NaN values with some value of their own. All these function help in filling a null values in datasets of a DataFrame. Interpolate() function is basically used to fill NA values in the dataframe but it uses various interpolation technique to fill the missing values rather than hard-coding the value.

Code #1: Filling null values with a single value

```
import pandas as pd
import numpy as np

# dictionary of lists
dict = {'First Score':[100, 90,np.nan,95],
        'Second Score': [30, 45,56,np.nan],
        'Third Score':[np.nan, 40,80,98]}

# creating a dataframe from dictionary
df = pd.DataFrame(dict)

# filling missing value using fillna()
df.fillna(0)
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	0.0
1	90.0	45.0	40.0
2	0.0	56.0	80.0
3	95.0	0.0	98.0

Code #2: Filling null values with the previous ones

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

Output

```
# dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score':[np.nan,40,80,np.nan]}

# creating a dataframe from dictionary
df = pd.DataFrame(dict)

# filling a missing value with
# previous ones
df.fillna(method ='pad')
```

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	90.0	56.0	80.0
3	95.0	56.0	80.0

Code #3: Filling null value with the next ones

```
import pandas as pd
import numpy as np

# dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from dictionary
df = pd.DataFrame(dict)

# filling  null value using fillna() function
df.fillna(method ='bfill')
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	40.0
1	90.0	45.0	40.0
2	95.0	56.0	80.0
3	95.0	NaN	98.0

Code #4: Filling null values in CSV File

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("employees.csv")

# Printing the first 10 to 24 rows of
# the data frame for visualization
Print(data[10:25])
```

This program prints the employee database starting from row 10 to Row 25. In the output, we can see Row 20 & Row 22, the values for gender is NaN

Output:

	First Name	Gender	Start Date	I
10	Louise	Female	8/12/1980	
11	Julia	Female	10/26/1997	
12	Brandon	Male	12/1/1980	
13	Gary	Male	1/27/2008	
14	Kimberly	Female	1/14/1999	
15	Lillian	Female	6/5/2016	
16	Jeremy	Male	9/21/2010	
17	Shawn	Male	12/7/1986	
18	Diana	Female	10/23/1981	
19	Donna	Female	7/22/2010	
20	Lois	NaN	4/22/1995	
21	Matthew	Male	9/5/1995	
22	Joshua	NaN	3/8/2012	
23	NaN	Male	6/14/2012	
24	John	Male	7/1/1992	

Now we are going to fill all the null values in Gender column with “No Gender”

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("employees.csv")
```

Output:

```
# filling a null values using fillna()
data["Gender"].fillna("No Gender",inplace=True)

print(data)
```

10	Louise	Female	8/12/1980
11	Julie	Female	10/26/1997
12	Brandon	Male	12/1/1980
13	Gary	Male	1/27/2008
14	Kimberly	Female	1/14/1999
15	Lillian	Female	6/5/2016
16	Jeremy	Male	9/21/2010
17	Shawn	Male	12/7/1986
18	Diana	Female	10/23/1981
19	Donna	Female	7/22/2010
20	Lois	No Gender	4/22/1995
21	Matthew	Male	9/5/1995
22	Joshua	No Gender	3/8/2012
23	NaN	Male	6/14/2012
24	John	Male	7/1/1992

Code #5: Filling a null values using replace() method

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("employees.csv")

# Printing the first 10 to 24 rows of
# the data frame for visualization
data[10:25]
```

Output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN

Now we are going to replace the all Nan value in the data frame with -99 value.

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("employees.csv")

# Replace  Nan value in dataframe with value -99
data.replace(to_replace = np.nan, value = -99)
```

Output:



Interpolation

Interpolation in Python is a technique used to estimate unknown data points between two known data points. Interpolation is mostly used to impute missing values in the data frame or series while pre-processing data. Interpolation is also used in Image Processing when expanding an image you can estimate the pixel value with help of neighbouring pixels.

When to use Interpolation?

We can use Interpolation to find missing value with help of its neighbours. When imputing missing values with average does not fit best, we have to move to a different technique and the technique most people find is Interpolation. Interpolation is mostly used while working with time-series data because in time-series data we like to fill missing values with previous one or two values. for example, suppose temperature, now we would always prefer to fill today's temperature with the mean of the last 2 days, not with the mean of the month. We can also use Interpolation for calculating the moving averages.

Using Interpolation to fill Missing Values in Series Data

Pandas series is a one-dimensional array which is capable to store elements of various data types like list. We can easily create series with help of a list, tuple, or dictionary. To perform all Interpolation methods we will create a pandas series with some NaN values and try to fill missing values with different methods of Interpolation.

```
import pandas as pd
import numpy as np
a = pd.Series([0, 1, np.nan, 3, 4, 5, 7],
              index=[100,101,102,103,104,105,106])
print(a)
```

Output:

```
100    0.0
101    1.0
102    NaN
103    3.0
104    4.0
105    5.0
106    7.0
dtype: float64
```

1) Linear Interpolation

Linear Interpolation simply means to estimate a missing value by connecting dots in a straight line in increasing order. In short, it estimates the unknown value in the same increasing order from previous values. The default method used by Interpolation is Linear so while applying it we did not need to specify it.

Code #6.1: Linear Interpolation

```
import pandas as pd
import numpy as np
a = pd.Series([0, 1, np.nan, 3, 4, 5, 7],
              index=[100,101,102,103,104,105,106])
print(a.interpolate())
```

Output:

```
100    0.0
101    1.0
102    2.0
103    3.0
104    4.0
105    5.0
106    7.0
dtype: float64
```

Hence, Linear interpolation works in the same order. Remember that it does not interpret using the index, it interprets values by connecting points in a straight line.

2) Polynomial Interpolation

In Polynomial Interpolation you need to specify an order. It means that polynomial interpolation is filling missing values with the lowest possible degree that passes through available data points. The polynomial Interpolation curve is like the trigonometric sin curve or assumes it like a parabola shape.

Code #6.2: Polynomial Interpolation

```
import pandas as pd
import numpy as np
a = pd.Series([0, 1, np.nan, 3, 4, 5, 7],
              index=[100,101,102,103,104,105,106])
a.interpolate(method="polynomial", order=2)
```

Output:

```
100    0.00000
101    1.00000
102    1.99537
103    3.00000
104    4.00000
105    5.00000
106    7.00000
dtype: float64
```

If you pass an order as 1 then the output will similar to linear because the polynomial of order 1 is linear.

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

Output:

```
a = pd.Series([0, 1, np.nan, 3, 4, 5, 7],
              index=[100,101,102,103,104,105,106])
a.interpolate(method="polynomial", order=1)
```

```
100    0.0
101    1.0
102    2.0
103    3.0
104    4.0
105    5.0
106    7.0
dtype: float64
```

3) Interpolation through Padding

Interpolation with help of padding simply means filling missing values with the same value present above them in the dataset. If the missing value is in the first row then this method will not work. While using this technique you also need to specify the limit which means how many NaN values to fill. So, if you are working on a real-world project and want to fill missing values with previous values you have to specify the limit as to the number of rows in the dataset.

Code #6.3: Padding Interpolation

```
import pandas as pd
import numpy as np
a = pd.Series([0, 1, np.nan, 3, 4, 5, 7],
              index=[100,101,102,103,104,105,106])
a.interpolate(method="pad", limit=2)
```

Output:

```
100    0.0
101    1.0
102    1.0
103    3.0
104    4.0
105    5.0
106    7.0
dtype: float64
```

```
import pandas as pd
import numpy as np
a = pd.Series([0, 1, np.nan, 3, 4, 5, 7],
              index=[100,101,102,103,104,105,106])
a.interpolate(method="bfill", limit=2)
```

Output:

```
100    0.0
101    1.0
102    3.0
103    3.0
104    4.0
105    5.0
106    7.0
dtype: float64
```

Using Interpolation to fill Missing Values in Pandas DataFrame

DataFrame is a widely used python data structure that stores the data in form of rows and columns. When performing data analysis we always store the data in a table which is known as a dataframe. Dataframe can contain huge missing values in many columns so let us understand how we can use Interpolation to fill missing values in the dataframe.

Code #7: Displaying the Data Frame

```
import pandas as pd
# Creating the dataframe
df = pd.DataFrame({"A":[12, 4, 7, None, 2],
                   "B":[None, 3, 57, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
print(df)
```

Output:

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	3.0	16.0	3.0
2	7.0	57.0	NaN	NaN
3	NaN	3.0	3.0	NaN
4	2.0	NaN	8.0	6.0

Displaying the Data Frame & Performing Linear Interpolation in forwarding Direction

The linear method ignores the index and treats missing values as equally spaced and finds the best point to fit the missing value after previous points. If the missing value is at first index then it will leave it as Nan. Let's apply it to our dataframe.

Code #7.1: Performing Linear Interpolation in forward Direction

```
import pandas as pd
# Creating the dataframe
df = pd.DataFrame({"A":[12, 4, 7, None, 2],
                   "B":[None, 3, 57, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
df.interpolate(method ='linear', limit_direction ='forward')
```

Output:

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	3.0	16.0	3.0
2	7.0	57.0	9.5	4.0
3	4.5	3.0	3.0	5.0
4	2.0	3.0	8.0	6.0

If you only want to perform interpolation in the single column then it is also simple and follows the below code.

```
import pandas as pd
# Creating the dataframe
df = pd.DataFrame({"A":[12, 4, 7, None, 2],
                   "B":[None, 3, 57, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
df['C'].interpolate(method="linear")
```

```
0    20.0
1    16.0
2     9.5
3     3.0
4     8.0
Name: C, dtype: floa
```

Interpolation with Padding

We have already seen that to use padding we have to specify the limit of NaN values to be filled. we have a maximum of 2 NaN values in the dataframe so our limit will be 2.

Code #7.2: Performing Interpolation with padding

```
import pandas as pd
# Creating the dataframe
df = pd.DataFrame({"A":[12, 4, 7, None, 2],
                   "B":[None, 3, 57, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
df.interpolate(method="pad", limit=2)
```

Output:

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	3.0	16.0	3.0
2	7.0	57.0	16.0	3.0
3	7.0	3.0	3.0	3.0
4	2.0	3.0	8.0	6.0

Hierarchical Indexing with Pandas

Till now we have focused primarily on one-dimensional and two-dimensional data, stored in Pandas Series and DataFrame objects, respectively. Often it is useful to go beyond this and store higher-dimensional data—that is, data indexed by more than one or two keys. While Pandas does provide Panel and Panel4D objects that natively handle three-dimensional and four-dimensional data, a far more common pattern in practice is to make use of hierarchical indexing (also known as multi-indexing) to incorporate multiple index levels within a single index.

```
#Importing libraries
import pandas as pd
import numpy as np
```

Output

```
data=pd.Series(np.random.randn(8),
               index=[["a","a","a","b","b","b","c","c"],
                      [1,2,3,1,2,3,1,2]])
print(data)
```

```
a  1    0.319500
   2   -0.727373
   3    0.951350
b  1    2.090693
   2   -0.781608
   3    0.431950
c  1    1.425975
   2    1.202417
dtype: float64
```

What is MultiIndex?

MultiIndex allows you to select more than one row and column in your index. To understand MultiIndex, let's see the indexes of the data.

```
#Importing libraries
import pandas as pd
import numpy as np
data=pd.Series(np.random.randn(8),
               index=[["a","a","a","b","b","b","c","c"],
                      [1,2,3,1,2,3,1,2]])
print(data.index)
```

Output

```
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 2),
            ('b', 3),
            ('c', 1),
            ('c', 2)],
           )
```

MultiIndex is an advanced indexing technique for DataFrames that shows the multiple levels of the indexes. Our dataset has two levels. You can obtain subsets of the data using the indexes. For example, let's take a look at the values with index a.

```
#Importing libraries
import pandas as pd
import numpy as np
data=pd.Series(np.random.randn(8),
               index=[["a","a","a","b","b","b","c","c"],
                      [1,2,3,1,2,3,1,2]])
print(data["a"])
```

Output

```
1   -1.788333
2   -0.555114
3   -1.793862
dtype: float64
```

```
#slicing can also be done on multiindexes
data["b":"c"]
```

Output

```
b  1    0.094040
   2   -0.558701
   3   -0.388170
c  1   -0.646243
   2    0.993541
dtype: float64
```

```
#We can also look more than one index
data.loc[["a","c"]]
```

Output

```
a  1    0.515388
   2   -0.630195
   3    0.169246
c  1    0.022259
   2    0.743997
dtype: float64
```

You can select values from the inner index. Let's take a look at the first values of the inner index.

```
#We can also look more than one index
data.loc[:,1]
```

Output

```
a    0.379618
b   -0.589347
c    2.173687
dtype: float64
```

What is the unstack?

The stack method turns column names into index values, and the unstack method turns index values into column names. You can see the data as a table with the unstack method.

```
#Importing libraries
import pandas as pd
import numpy as np
data = pd.Series(np.random.randn(8),
                 index=[["a","a","a","b","b","b","c","c"]
,
                        [1,2,3,1,2,3,1,2]])
data.unstack()
```

Output

	1	2	3
a	0.228471	0.703303	-0.314696
b	-0.544562	-0.861732	-1.874380
c	-0.134512	0.262817	NaN

To restore the dataset, you can use the stack method.

```
data.unstack().stack()
```

Output

```
a  1   -0.026588
   2   -0.629344
   3    1.857649
b  1   -2.651214
   2    0.986683
   3   -1.061997
c  1    1.376184
   2    0.322362
dtype: float64
```

Hierarchical Indexing in The Data Frame

You can move the DataFrame's columns to the row index. To show this, let's create a dataset.

```
#Importing libraries
import pandas as pd
import numpy as np
data=pd.DataFrame({"x":range(8),
                   "y":range(8,0,-1),
                   "a":["one","one","one","one","two","two","two","two"],
                   "b":[30,41,22,33,5,133,21,31]})
print(data)
```

Output:

```
   x  y    a    b
0  0  8  one   30
1  1  7  one   41
2  2  6  one   22
3  3  5  one   33
4  4  4  two    5
5  5  3  two  133
6  6  2  two   21
7  7  1  two   31
```

```
data2=data.set_index(["a","b"])
data2
```

Output:

		x	y
a	b		
one	30	0	8
	41	1	7
	22	2	6
	33	3	5
two	5	4	4
	133	5	3
	21	6	2
	31	7	1

```
data3=data.set_index(["a","b"]).sort_index()
data3
```

Output:

		x	y
a	b		
one	22	2	6
	30	0	8
	33	3	5
	41	1	7
two	5	4	4
	21	6	2
	31	7	1
	133	5	3

APEC

Combining Data in Pandas with append(), merge(), join(), and concat()

pandas concat(): Combining Data Across Rows or Columns

Concatenation is a bit different from the merging techniques that you saw above. With merging, you can expect the resulting dataset to have rows from the parent datasets mixed in together, often based on some commonality.

With concatenation, your datasets are just stitched together along an axis — either the row axis or column axis.

```
#Importing libraries
import pandas as pd
import numpy as np

df1 = pd.DataFrame(
    {
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    },
    index=[0, 1, 2, 3],
)
df2 = pd.DataFrame(
    {
        "A": ["A4", "A5", "A6", "A7"],
        "B": ["B4", "B5", "B6", "B7"],
        "C": ["C4", "C5", "C6", "C7"],
        "D": ["D4", "D5", "D6", "D7"],
    },
    index=[4, 5, 6, 7],
)
df3 = pd.DataFrame(
    {
        "A": ["A8", "A9", "A10", "A11"],
        "B": ["B8", "B9", "B10", "B11"],
        "C": ["C8", "C9", "C10", "C11"],
        "D": ["D8", "D9", "D10", "D11"],
    },
    index=[8, 9, 10, 11],
)
frames = [df1, df2, df3]
result = pd.concat(frames)
print("\n", df1)
print("\n", df2)
print("\n", df3)
print("\n", result)
```

Output:

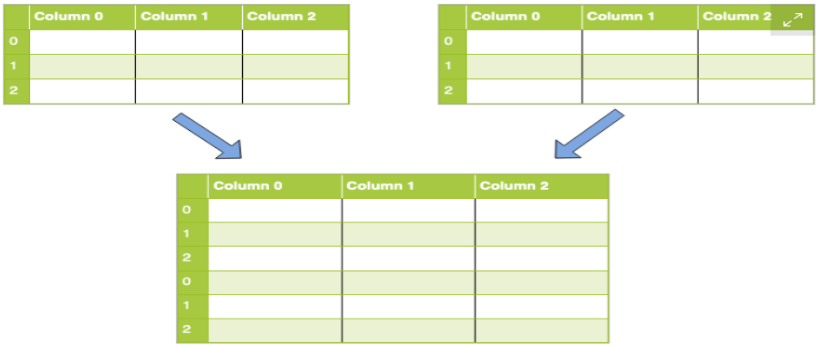
```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3

    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7

     A    B    C    D
8   A8   B8   C8   D8
9   A9   B9   C9   D9
10  A10  B10  C10  D10
11  A11  B11  C11  D11

     A    B    C    D
0   A0   B0   C0   D0
1   A1   B1   C1   D1
2   A2   B2   C2   D2
3   A3   B3   C3   D3
4   A4   B4   C4   D4
5   A5   B5   C5   D5
6   A6   B6   C6   D6
7   A7   B7   C7   D7
8   A8   B8   C8   D8
9   A9   B9   C9   D9
10  A10  B10  C10  D10
11  A11  B11  C11  D11
```

Visually, a concatenation with no parameters along rows would look like this:




```
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1"],
        "B": ["B0", "B1"],
        "C": ["C0", "C1"],
        "D": ["D0", "D1"],
    },index=[0, 1])
df2 = pd.DataFrame(
    {
        "A": ["A4", "A5"],
        "B": ["B4", "B5"],
        "C": ["C4", "C5"],
        "D": ["D4", "D5"],
    },index=[0, 1])
frames = [df1, df2]
result = pd.concat(frames)
print("\n", df1)
print("\n", df2)
print("\n", result)
```

Output:

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1

    A   B   C   D
0  A4  B4  C4  D4
1  A5  B5  C5  D5

    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
0  A4  B4  C4  D4
1  A5  B5  C5  D5
```

result = pd.concat((frames), axis = "rows")

```
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1"],
        "B": ["B0", "B1"],
        "C": ["C0", "C1"],
        "D": ["D0", "D1"],
    },index=[0, 1])
df2 = pd.DataFrame(
    {
        "A": ["A4", "A5"],
        "B": ["B4", "B5"],
        "C": ["C4", "C5"],
        "D": ["D4", "D5"],
    },index=[0, 1])
frames = [df1, df2]
result = pd.concat((frames),
                   axis = "columns")
print("\n", df1)
print("\n", df2)
print("\n", result)
```

Output:

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1

    A   B   C   D
0  A4  B4  C4  D4
1  A5  B5  C5  D5

    A   B   C   D   A   B   C   D
0  A0  B0  C0  D0  A4  B4  C4  D4
1  A1  B1  C1  D1  A5  B5  C5  D5
```

```
result = pd.concat(frames, keys=["x", "y"])
```

As you can see, the resulting object's index has a hierarchical

Output:

index.	

```
      A   B   C   D
x 0  A0  B0  C0  D0
  1  A1  B1  C1  D1
y 0  A4  B4  C4  D4
  1  A5  B5  C5  D5
```

APEC

In Pandas for a horizontal combination we have merge() and join(), whereas for vertical combination we can use concat() and append(). Merge and join perform similar tasks but internally they have some differences, similar to concat and append.

pandas merge():

Pandas provides various built-in functions for easily combining datasets. Among them, merge() is a high-performance in-memory operation very similar to relational databases like SQL. You can use merge() any time when you want to do database-like join operations.

- The simplest call without any key column
- Specifying key columns using **on**
- Merging using left_on and right_on
- Various forms of joins: inner, left, right and outer

Syntax:

- # This join brings together the entire DataFame
  df.merge(df2)

- # This join only brings together a subset of columns
- # 'col1' is my key in this case

  df[['col1', 'col2']].merge(df2[['col1', 'col3']], on='col1')

Code 1#: Merging two DataFrames

```
df1 = pd.DataFrame({
    'id': [1, 2, 3, 4],
    'name': ['Tom', 'Jenny', 'James', 'Dan'],
})
df2 = pd.DataFrame({
    'id': [2, 3, 4, 5],
    'age': [31, 20, 40, 70],
    'sex': ['F', 'M', 'M', 'F']
})
print("\n",df1)
print("\n",df2)

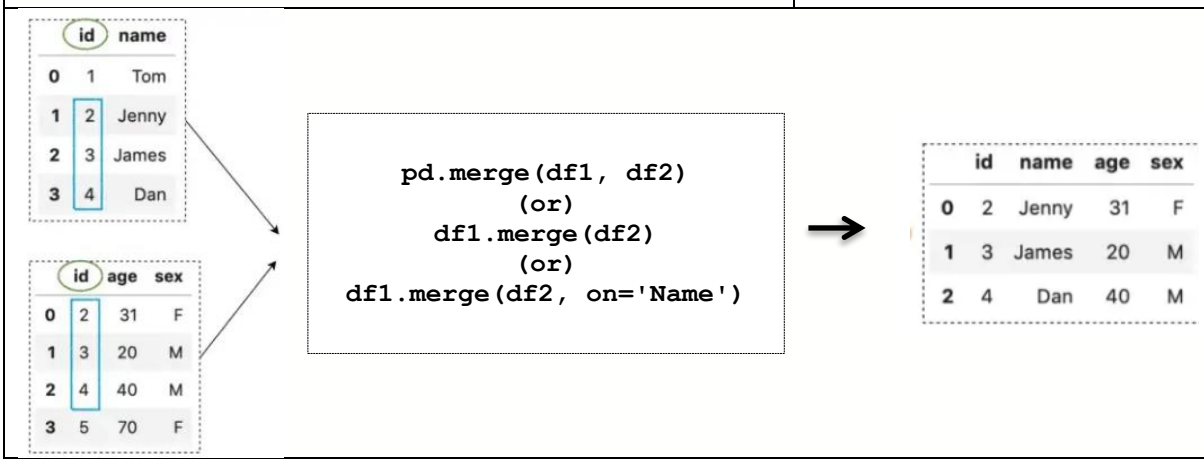
final = pd.merge(df1, df2)
print("\n",final)
```

Output:

```
    id   name
0   1    Tom
1   2  Jenny
2   3  James
3   4    Dan

    id  age sex
0   2   31   F
1   3   20   M
2   4   40   M
3   5   70   F

    id   name  age sex
0   2  Jenny   31   F
1   3  James   20   M
2   4    Dan   40   M
```

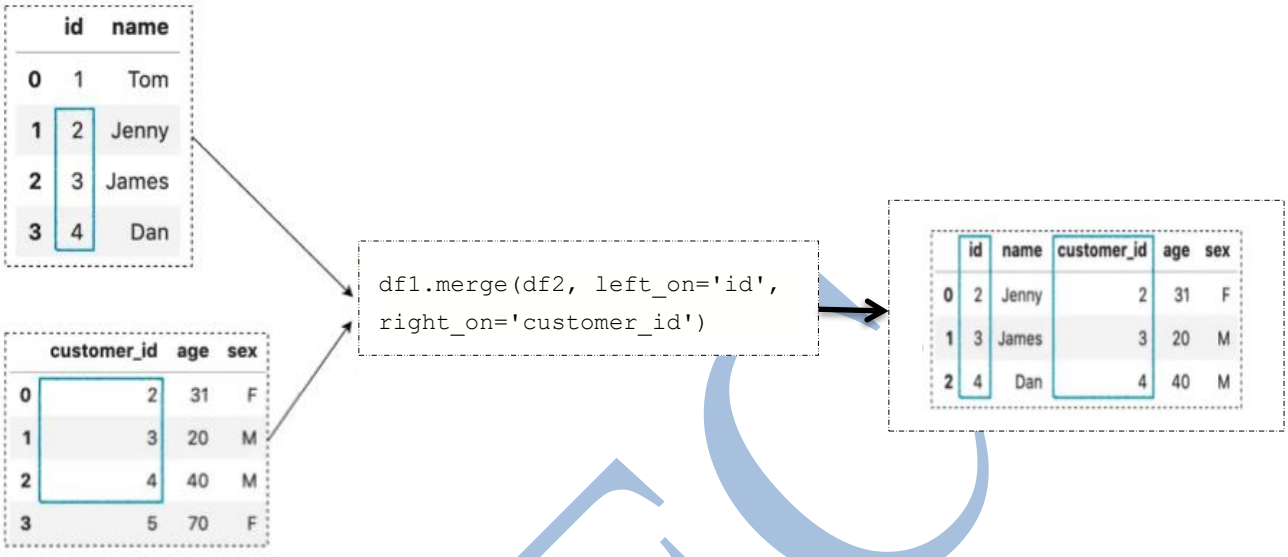


Code 2#: Merge two DataFrames via 'id' column.

```
final = df1.merge(df2, on='id')
print("\n",final)
```

Output:

```
   id   name  age sex
0   2  Jenny   31   F
1   3  James   20   M
2   4    Dan   40   M
```



Code 3#: Merge with different column names - specify a left_on and right_on

```
final = df1.merge(df2, left_on='id',
                  right_on='customer_id')
```

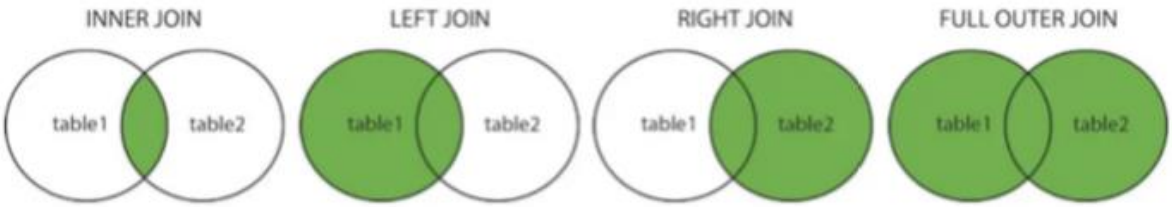
Output:

```
   id   name  customer_id  age sex
0   2  Jenny            2   31   F
1   3  James            3   20   M
2   4    Dan            4   40   M
```

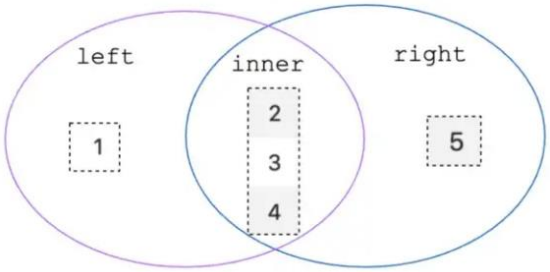
Various type of joins: inner, left, right and outer

They are 4 types of joins available to Pandas merge() function. The logic behind these joins is very much the same that you have in SQL when you join tables. You can perform a type of join by specifying the how argument with the following values:

- **inner:** Default join is inner in Pandas merge() function, and it produces records that have matching values in both DataFrames
- **left:** produces all records from the left DataFrame and the matched records from the right DataFrame
- **right:** produces all records from the right DataFrame and the matched records from the left DataFrame
- **outer:** produces all records when there is a match in either left or right DataFrame




```
pd.merge(df_customer, df_info, on='id', how=?)
```



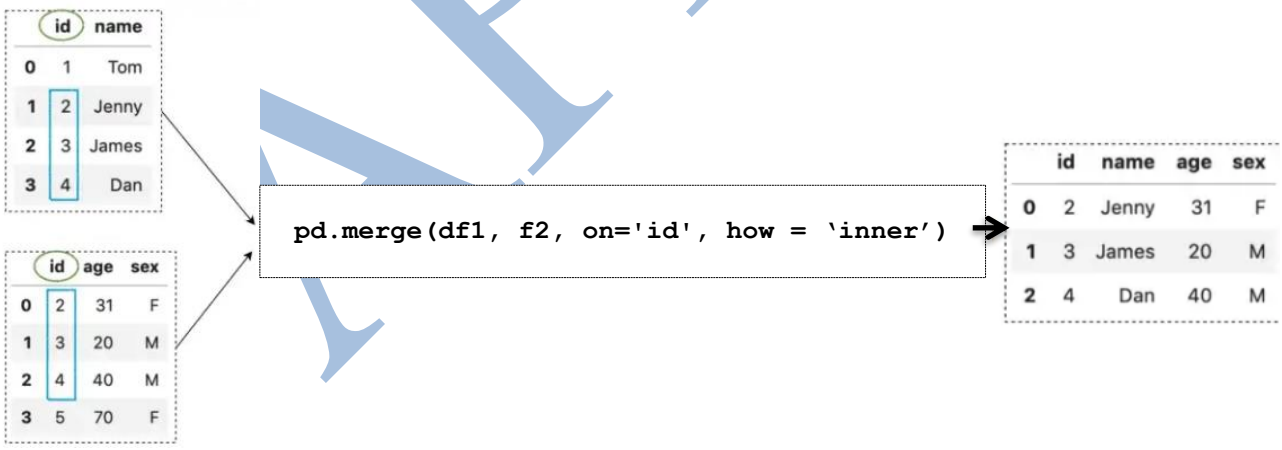
Code 4#: Merge using inner join

Pandas merge() is performing the inner join and it produces only the set of records that match in both DataFrame.

```
final = pd.merge(df1, df2, on='id',
                 how = 'inner')
```

Output:

```
   id    name   age sex
0   2   Jenny    31   F
1   3   James    20   M
2   4     Dan    40   M
```



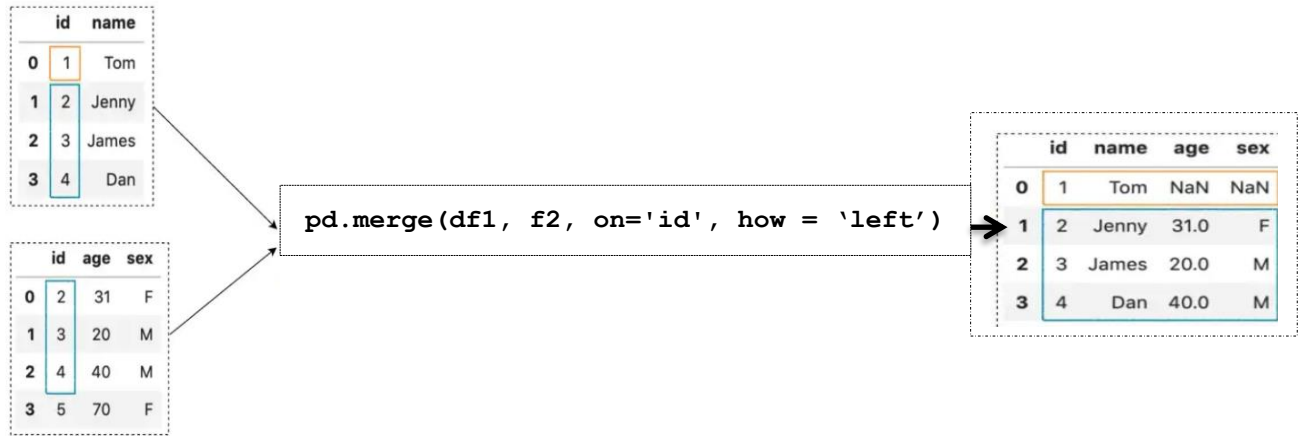
Code 4#: Merge using Left join

The left join produces all records from the left DataFrame, and the matched records from the right DataFrame. If there is no match, the left side will contain NaN.

```
final = pd.merge(df1, df2, on='id',
                 how = 'left')
```

Output:

```
     id    name    age  sex
0    1     Tom    NaN  NaN
1    2   Jenny   31.0    F
2    3   James   20.0    M
3    4     Dan   40.0    M
```



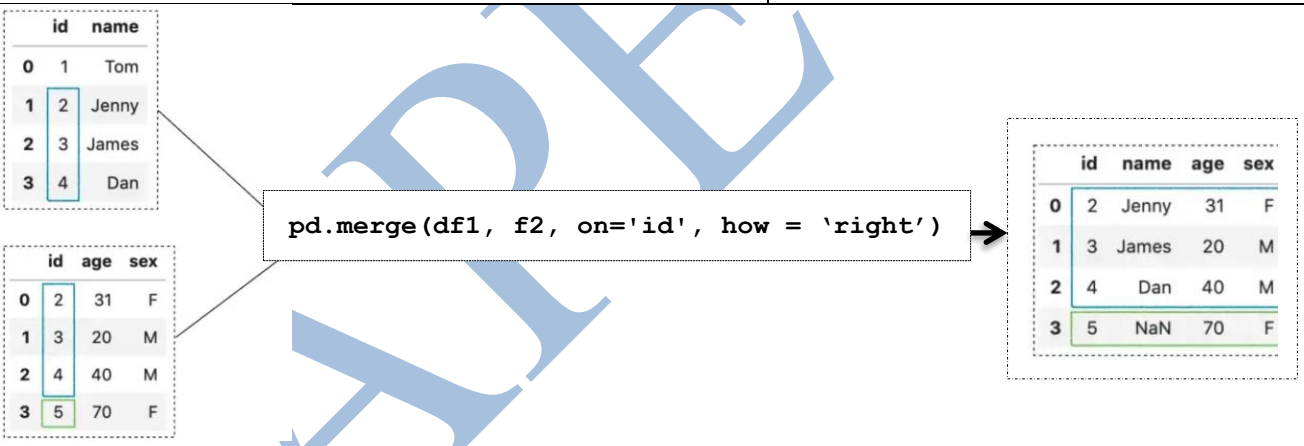
Code 4#: Merge using Right join

The right join produces all records from the right DataFrame, and the matched records from the left DataFrame. If there is no match, the right side will contain NaN.

```
final = pd.merge(df1, df2, on='id',
                 how = 'right')
```

Output:

```
   id   name  age sex
0   2  Jenny   31   F
1   3  James   20   M
2   4    Dan   40   M
3   5    NaN   70   F
```



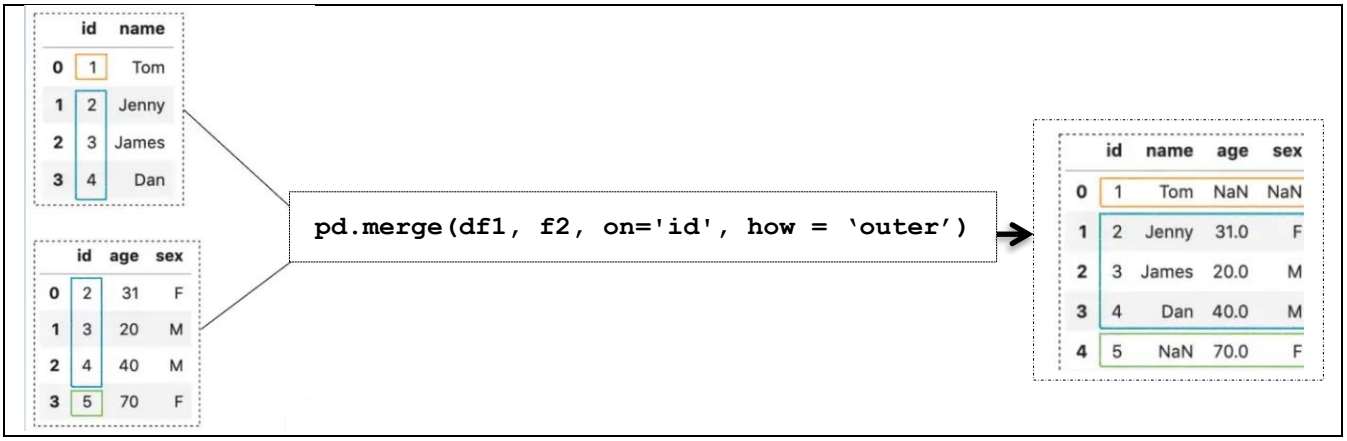
Code 4#: Merge using Outer join

The outer join produces all records when there is a match in either left or right DataFrame. NaN will be filled for no match on either sides.

```
final = pd.merge(df1, df2, on='id',
                 how = 'outer')
```

Output:

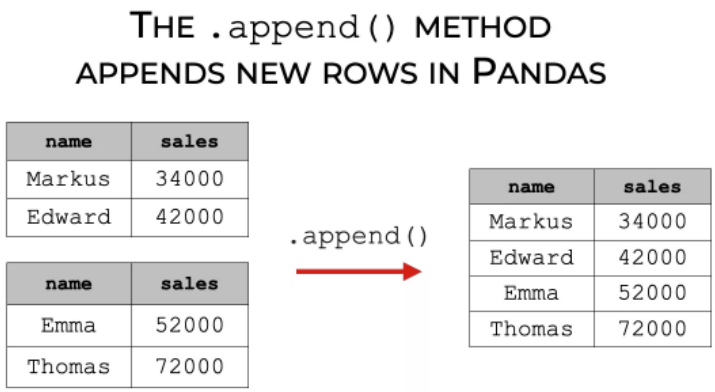
```
   id   name   age  sex
0   1    Tom   NaN  NaN
1   2  Jenny  31.0    F
2   3  James  20.0    M
3   4    Dan  40.0    M
4   5    NaN  70.0    F
```


id name
0 1 Tom
1 2 Jenny
2 3 James
3 4 Dan
id age sex
0 2 31 F
1 3 20 M
2 4 40 M
3 5 70 F
pd.merge(df1, f2, on='id', how = 'outer')
id name age sex
0 1 Tom NaN NaN
1 2 Jenny 31.0 F
2 3 James 20.0 M
3 4 Dan 40.0 M
4 5 NaN 70.0 F

APEC

pandas append():

To append the rows of one dataframe with the rows of another, we can use the Pandas append() function. With the help of append(), we can append columns too.



Steps

- Create a two-dimensional, size-mutable, potentially heterogeneous tabular data, df1.
- Print the input DataFrame, df1.
- Create another DataFrame, df2, with the same column names and print it.
- Use the append method, df1.append(df2, ignore_index=True), to append the rows of df2 with df2.
- Print the resultatnt DataFrame.

Code 5#: Append Function to join DataFrames

```
import pandas as pd
df1 = pd.DataFrame({"x": [5, 2],
                    "y": [4, 7],
                    "z": [1, 3]})
df2 = pd.DataFrame({"x": [1, 3],
                    "y": [1, 9],
                    "z": [1, 3]})
print ("\n", df1)
print ("\n", df2)
df3 = df1.append(df2)
print ("\n ", df3)
```

Output:

```
   x  y  z
0  5  4  1
1  2  7  3

   x  y  z
0  1  1  1
1  3  9  3

   x  y  z
0  5  4  1
1  2  7  3
0  1  1  1
1  3  9  3
```

```
df3 = df1.append(df2,ignore_index=True)
```

Output:

```
   x  y  z
0  5  4  1
1  2  7  3
2  1  1  1
3  3  9  3
```

```
import pandas as pd
df1 = pd.DataFrame({"x": [5, 2],
                    "y": [4, 7],
                    "z": [1, 3]})
df2 = pd.DataFrame({"a": [1, 3],
                    "b": [1, 9],
                    "c": [1, 3]})
print ("\n", df1)
print ("\n", df2)
df3 = df1.append(df2, ignore_index=True)
```

Output:

```
print ("\n ", df3)
```

```
   x  y  z
0  5  4  1
1  2  7  3

   a  b  c
0  1  1  1
1  3  9  3

     x    y    z    a    b    c
0  5.0  4.0  1.0  NaN  NaN  NaN
1  2.0  7.0  3.0  NaN  NaN  NaN
2  NaN  NaN  NaN  1.0  1.0  1.0
3  NaN  NaN  NaN  3.0  9.0  3.0
```

Aggregation in Pandas

Aggregation in pandas provides various functions that perform a mathematical or logical operation on our dataset and returns a summary of that function. Aggregation can be used to get a summary of columns in our dataset like getting sum, minimum, maximum, etc. from a particular column of our dataset.

Some functions used in the aggregation are:

- sum() Compute sum of column values
- min() Compute min of column values
- max() Compute max of column values
- mean() Compute mean of column
- size() Compute column sizes
- describe() Generates descriptive statistics
- first() Compute first of group values
- last() Compute last of group values
- count() Compute count of column values
- std() Standard deviation of column
- var() Compute variance of column
- sem() Standard error of the mean of column
-

```
df = pd.DataFrame([[9, 4, 8, 9],
                   [8, 10, 7, 6],
                   [7, 6, 8, 5]],
                  columns=['Maths',  'English',
                           'Science', 'History'])

# display dataset
print(df)
```

Output

```
   Maths  English  Science  History
0      9        4        8        9
1      8       10        7        6
2      7        6        8        5
```

agg() Calculate the sum, min, and max of each column in our dataset.

```
df.agg(['sum', 'min', 'max'])
```

	Maths	English	Science	History
sum	24	20	23	20
min	7	4	7	5
max	9	10	8	9

APEC

Grouping in Pandas

Grouping is used to group data using some criteria from our dataset. It is used as split-apply-combine strategy.

- Splitting the data into groups based on some criteria.
- Applying a function to each group independently.
- Combining the results into a data structure.

Examples:

- We use groupby() function to group the data on "Maths" value. It returns the object as result.

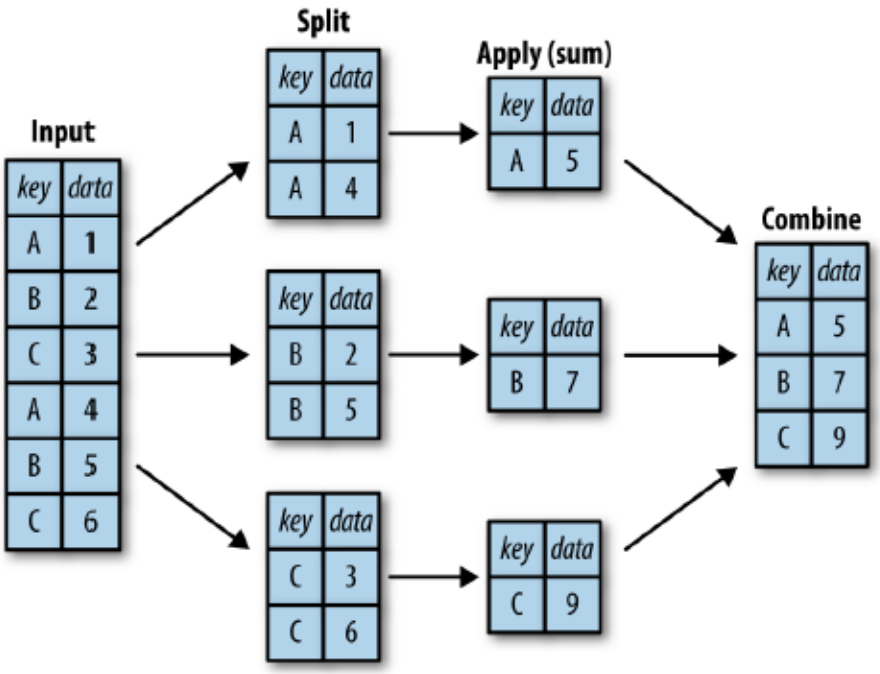
```
df.groupby(by=['Maths'])
```

Applying groupby() function to group the data on "Maths" value. To view result of formed groups use first() function.

```
a = df.groupby('Maths')
a.first()
```

First grouping based on "Maths" within each team we are grouping based on "Science"

```
b = df.groupby(['Maths', 'Science'])
b.first()
```



Simple DataFrame

```
import pandas as pd
df = pd.DataFrame(
    {
        "A": [1, 1, 2, 2],
        "B": [1, 2, 3, 4],
        "C": [0.362838,0.227877,1.267767,-0.562860]
    }
)
print(df)
```

Output:

```
   A  B         C
0  1  1  0.362838
1  1  2  0.227877
2  2  3  1.267767
3  2  4 -0.562860
```

The aggregation is for each column.

```
df.groupby('A').agg('min')
```

```
   B         C
A
```

	1 1 0.227877 2 3 -0.562860
Multiple aggregations `df.groupby('A').agg(['min', 'max'])`	B C min max min max A 1 1 2 0.227877 0.362838 2 3 4 -0.562860 1.267767
Select a column for aggregation `df.groupby('A').B.agg(['min', 'max'])`	min max A 1 1 2 2 3 4
Different aggregations per column `df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})`	B C min max sum A 1 1 2 0.590715 2 3 4 0.704907

`df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],`
`'data': range(6)}, columns=['key', 'data'])`
`print(df)`
	key data
	0 A 0
	1 B 1
	2 C 2
	3 A 3
	4 B 4
	5 C 5

`df.groupby('key').sum())`	data key A 3 B 5 C 7
Transformation. While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input.	
`df.sum()`	key ABCABC data 15 dtype: object
`df.mean()`	data 2.5
`df.groupby('key').transform(lambda x: x - x.mean())`	data 0 -1.5 1 -1.5 2 -1.5 3 1.5 4 1.5 5 1.5

Pivot Tables

We have seen how the GroupBy abstraction lets us explore relationships within a dataset. A pivot table is a similar operation that is commonly seen in spread sheets and other programs that operate on tabular data. The pivot table takes simple column wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data. The difference between pivot tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a multidimensional version of GroupBy aggregation.

That is, you split apply- combine, but both the split and the combine happen across not a one dimensional index, but across a two-dimensional grid.

Purpose:

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table of pandas will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

How to make a pivot table?

Use the pd.pivot_table() function and specify what feature should go in the rows and columns using the index and columns parameters respectively. The feature that should be used to fill in the cell values should be specified in the values parameter.

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'First Name':['Aryan','Rohan','Riya','Yash','Siddhant',],
                   'Last Name:['Singh','Agarwal','Shah','Bhatia','Khanna'],
                   'Type:['Full-time Employee','Intern','Full-time Employee',
                            'Part-time Employee', 'Full-time Employee'],
                   'Department': ['Administration','Technical','Administration',
                                  'Technical','Management'],
                   'YoE': [2, 3, 5, 7, 6],
                   'Salary': [20000, 5000, 10000, 10000, 20000]})

Print(df)
```

Output:

	First Name	Last Name	Type	Department	YoE	Salary
0	Aryan	Singh	Full-time Employee	Administration	2	20000
1	Rohan	Agarwal	Intern	Technical	3	5000
2	Riya	Shah	Full-time Employee	Administration	5	10000
3	Yash	Bhatia	Part-time Employee	Technical	7	10000
4	Siddhant	Khanna	Full-time Employee	Management	6	20000

```
output = pd.pivot_table(data=df,
                        index=['Type'],
                        columns=['Department'],
                        values='Salary',
                        aggfunc='mean')
print(output)
```

Department	Administration	Management	Technical
Type			
Full-time Employee	15000.0	20000.0	NaN
Intern	NaN	NaN	5000.0
Part-time Employee	NaN	NaN	10000.0

Here, we have made a basic pivot table in pandas which shows the average salary of each type of employee for each department. As there are no user-defined parameters passed, the remaining arguments have assumed their default values.

Pivot table with multiple aggregation functions

```
Df1 = pd.pivot_table(data=df, index=['Type'],
                         values='Salary',
                         aggfunc=['sum', 'mean', 'count'])
print(df1)
```

Output

	sum	mean	count
	Salary	Salary	Salary
Type			
Full-time Employee	50000	16666.666667	3
Intern	5000	5000.000000	1
Part-time Employee	10000	10000.000000	1

Calculating row and column grand totals in pivot table

Now, let's take a look at the grand total of the salary of each type of employee. For this, we will use the margins and the margins_name parameter.

```
# Calculate row and column totals (margins)
df1 = pd.pivot_table(data=df, index=['Type'],
                         values='Salary',
                         aggfunc=['sum', 'mean', 'count'],
                         margins=True,
                         margins_name='Grand Total')
print(df1)
```

```
                      sum          mean  count
                   Salary        Salary Salary
Type
Full-time Employee  50000  16666.666667      3
Intern               5000   5000.000000      1
Part-time Employee  10000  10000.000000      1
Grand Total         65000  13000.000000      5
```

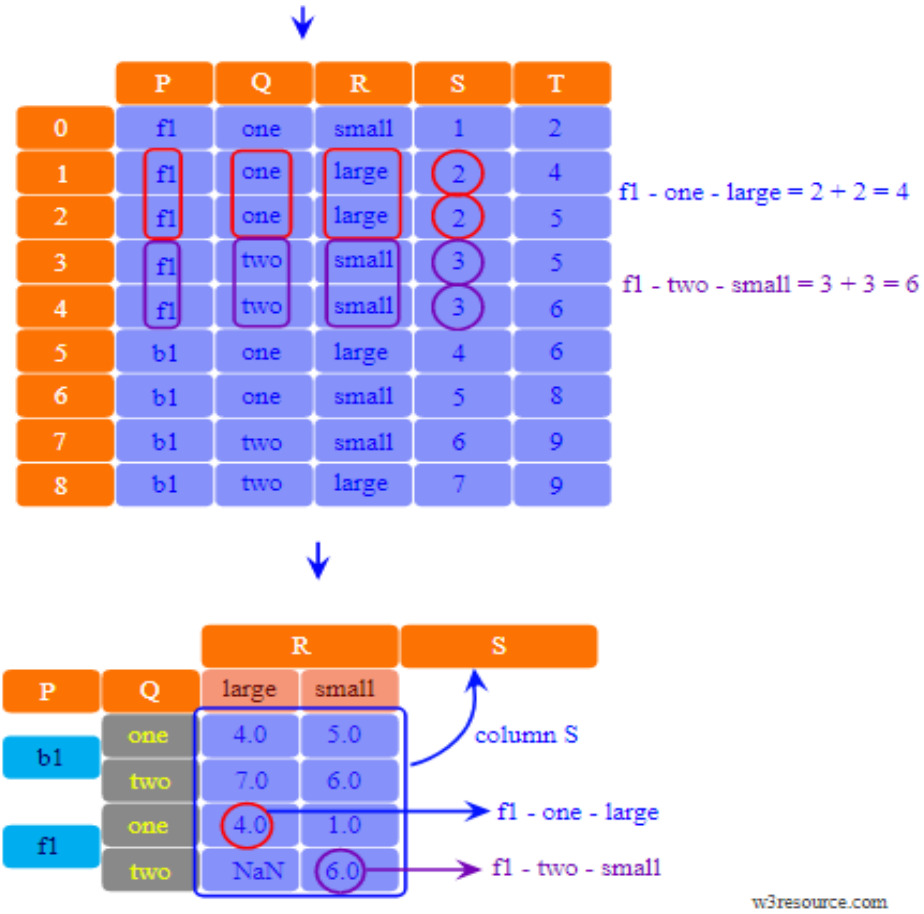
```
df = pd.DataFrame({"P": ["f1","f1","f1","f1","f1","b1","b1","b1","b1"],
                   "Q": ["one", "one", "one", "two", "two",
                         "one", "one", "two", "two"],
                   "R": ["small", "large", "large", "small",
                         "small", "large", "small", "small",
                         "large"],
```

```
                    "S": [1, 2, 2, 3, 3, 4, 5, 6, 7],
                    "T": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
print(df)
```

Output

```
    P    Q      R  S  T
0  f1  one  small  1  2
1  f1  one  large  2  4
2  f1  one  large  2  5
3  f1  two  small  3  5
4  f1  two  small  3  6
5  b1  one  large  4  6
6  b1  one  small  5  8
7  b1  two  small  6  9
8  b1  two  large  7  9
```

```
table = pd.pivot_table(df, values='S', index=['P', 'Q'],
                       columns=['R'], aggfunc=np.sum)
print(table)
```



Output:

	R	large	small
P	Q		
b1	one	4.0	5.0
	two	7.0	6.0
f1	one	4.0	1.0
	two	NaN	6.0

```
table = pd.pivot_table(df, values='S', index=['P', 'Q'],
                       columns=['R'], aggfunc=np.sum, fill_value=9999)
print(table)
```

	R	large	small
P	Q		
b1	one	4	5
	two	7	6
f1	one	4	1
	two	9999	6

```
df.sort_values(by=['R'])
```

	P	Q	R	S	T
1	f1	one	large	2	4
2	f1	one	large	2	5
5	b1	one	large	4	6
8	b1	two	large	7	9
0	f1	one	small	1	2
3	f1	two	small	3	5
4	f1	two	small	3	6
6	b1	one	small	5	8
7	b1	two	small	6	9

Vectorized Strings

Vectorized string operations refer to performing operations on strings in a vectorized manner, meaning that the operations are applied simultaneously to multiple strings rather than individually. This approach leverages the power of optimized low-level operations provided by modern programming languages and libraries. In many programming languages, vectorized string operations are supported through libraries or modules that provide efficient string handling functions.

For example, in Python, the NumPy and pandas libraries offer vectorized string operations. Here are a few examples of vectorized string operations commonly used in programming languages:

- Concatenation: Joining multiple strings together. For instance, given two arrays of strings, a vectorized operation can concatenate the corresponding strings in each array to create a new array.
- Substring extraction: Extracting a portion of a string based on a specified start and end index. Vectorized substring extraction can be performed on multiple strings simultaneously.
- Case conversion: Changing the case of strings, such as converting all characters to uppercase or lowercase. Vectorized operations can be used to apply case conversion to multiple strings at once.
- String matching: Finding strings that match a specific pattern or regular expression. Vectorized string matching allows searching for matches across multiple strings efficiently.
- Replacement: Replacing substrings or patterns within strings. Vectorized replacement operations can replace substrings across multiple strings simultaneously.

This vectorization of operations simplifies the syntax of operating on arrays of data

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
print(x)
```

Output

```
[ 2  3  5  7 11 13]
```

```
print(x*2)
```

```
[ 4  6 10 14 22 26]
```

For arrays of strings, NumPy does not provide such simple access

```
data=['peter','Paul','MARY','gUIDO']
print(data)
```

Output

```
['peter','Paul','MARY','gUIDO']
```

```
data=['peter','Paul','MARY','gUIDO']
[s.capitalize() for s in data]
```

```
['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

```
data = ['peter', 'Paul', none, 'MARY', 'gUIDO']
[s.capitalize() for s in data]

NameError                                 Traceback (most recent call last)
<ipython-input-4-6d5ec95d781b> in <cell line: 1>()
----> 1 data = ['peter', 'Paul', none, 'MARY', 'gUIDO']
      2 [s.capitalize() for s in data]

NameError: name 'none' is not defined
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the `str` attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

```
import pandas as pd
names = pd.Series(data)
print(names)
```

Output

```
0    peter
1     Paul
2     none
3     MARY
4    gUIDO
```

	dtype: object
We can now call a single method that will capitalize all the entries, while skipping over any missing values:	
import pandas as pd names = pd.Series(data) names.str.capitalize()	**Output** 0 Peter 1 Paul 2 None 3 Mary 4 Guido dtype: object

Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas str methods that mirror Python string methods:

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()
zfill()	index()	isalpha()	split()
strip()	rindex()	isdigit()	rsplit()
rstrip()	capitalize()	isspace()	partition()
lstrip()	swapcase()	istitle()	rpartition()

names.str.lower()	0 peter 1 paul 2 none 3 mary 4 guido dtype: object
names.str.len()	0 5 1 4 2 4 3 4 4 5 dtype: int64
names.str.startswith('M')	0 False 1 False 2 False 3 True 4 False dtype: bool
import pandas as pd data = ['peter Charles', 'Paul Roudridge', 'MARY Siva', 'gUIDO'] names = pd.Series(data) names.str.split()	0 [peter, Charles] 1 [Paul, Roudridge] 2 [MARY, Siva] 3 [gUIDO] dtype: object

Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in re module

Mapping between Pandas methods and functions in Python's re module

Method	Description
match()	Call re.match() on each element, returning a Boolean.
extract()	Call re.match() on each element, returning matched groups as strings.
findall()	Call re.findall() on each element.
replace()	Replace occurrences of pattern with some other string.
contains()	Call re.search() on each element, returning a Boolean.
count()	Count occurrences of pattern.
split()	Equivalent to str.split(), but accepts regexps.
rsplit()	Equivalent to str.rsplit(), but accepts regexps.

Extract The First Name

```
names.str.extract('([A-Za-z]+)')
```

```
0    peter
1     Paul
2     MARY
3     gUIDO
```

Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations.

Other Pandas string methods

Method	Description
get()	Index each element
slice()	Slice each element
slice_replace()	Replace slice in each element with passed value
cat()	Concatenate strings
repeat()	Repeat values
normalize()	Return Unicode form of string
pad()	Add whitespace to left, right, or both sides of strings
wrap()	Split long strings into lines with length less than a given width
join()	Join strings in each element of the Series with passed separator
get_dummies()	Extract dummy variables as a DataFrame

Vectorized item access and slicing. The get() and slice() operations enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using str.slice(0, 3).

```
df.str.slice(0, 3) is equivalent to df.str[0:3]:
```

```
names.str[0:3]
```

```
0    pet
1    Pau
2    MAR
```

	3 gUI dtype: object
names.str.split().str.get(-1)	0 Charles 1 Roudridge 2 Siva 3 gUIDO dtype: object

APEC