UNIT IV
PYTHON LIBRARIES FOR DATA WRANGLING

Basics of Numpy arrays –aggregations –computations on arrays – comparisons, masks, boolean logic – fancy indexing – structured arrays – Data manipulation with Pandas – data indexing and selection – operating on data – missing data – Hierarchical indexing – combining datasets – aggregation and grouping – pivot tables

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 import numpy as np s = pd.Series(5, index=[0, 1, 2, 3]) print (s)	0 5 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

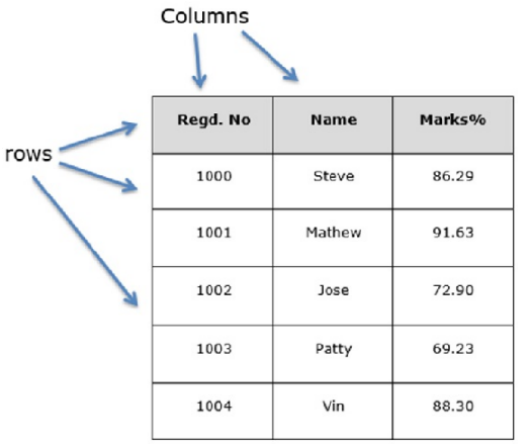
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

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
```

```
Given Data Frame:
   one  two
```

```
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)

print (df.loc['b'])
```

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

one   22
two    2
Name: b, dtype: int64
```

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

```
one     3
two   300
```

APEC

'two' : pd.Series([100, 200, 300], index=['a', 'b', 'c'])} df = pd.DataFrame(d) print (df.iloc[2])	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])
}
 	
#Create a DataFrame
df = pd.DataFrame(d)
print (df)
	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
	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])}
```

```
Describing various datas.....
Age
count  12.000000
mean   31.833333
std     9.232682
min    23.000000
```

APEC

#Create a DataFrame
df = pd.DataFrame(d)
print("Describing various datas.....")
print (df.describe())
	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	Our Data Frame: **Name Age Rating** 0 Tom 25 4.23 1 James 26 3.24

```
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))

# 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))
```

```
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:

```
   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:

APEC

	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')
print(df)
```

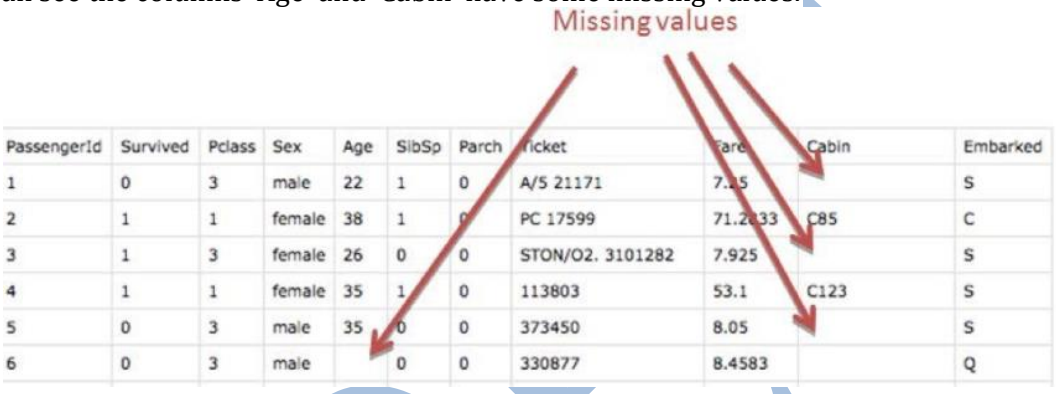
APEC

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.



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)

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

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.

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

Output

Data Frame:

```
# 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')
```

```
   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

# 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)
```

Output

	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

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

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
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	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")

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

print(data)
```

Output:

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	8/14/2012	4:19 PM	125792	5.042	NaN	NaN

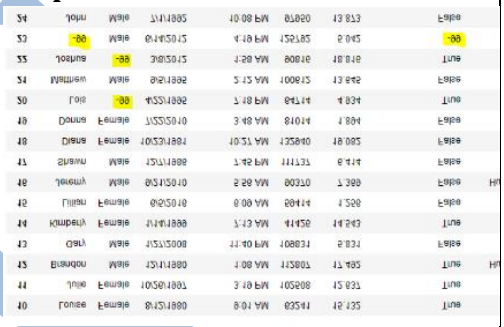
Now we are going to replace the all Nan value in the data frame with -99 value.

```
# 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
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)
```

Output:

```
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],
```

Output:

```
                index=[100,101,102,103,104,105,106])
a.interpolate(method="pad", limit=2)
```

```
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
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)
```

Output

```
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]])
```

Output

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

data.unstack()	
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:

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

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

		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:

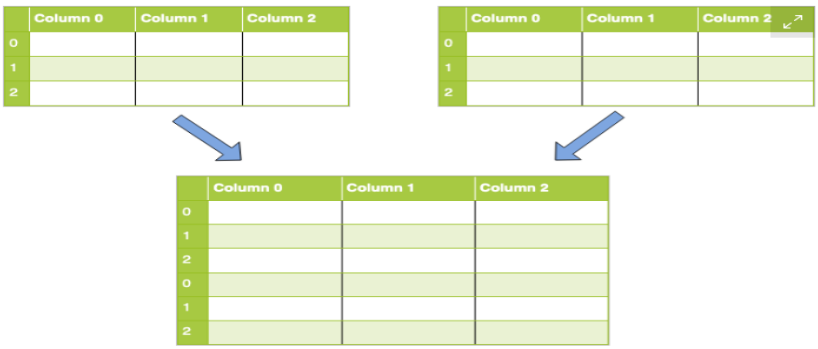
```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3

    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 index.

Output:

```
      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)

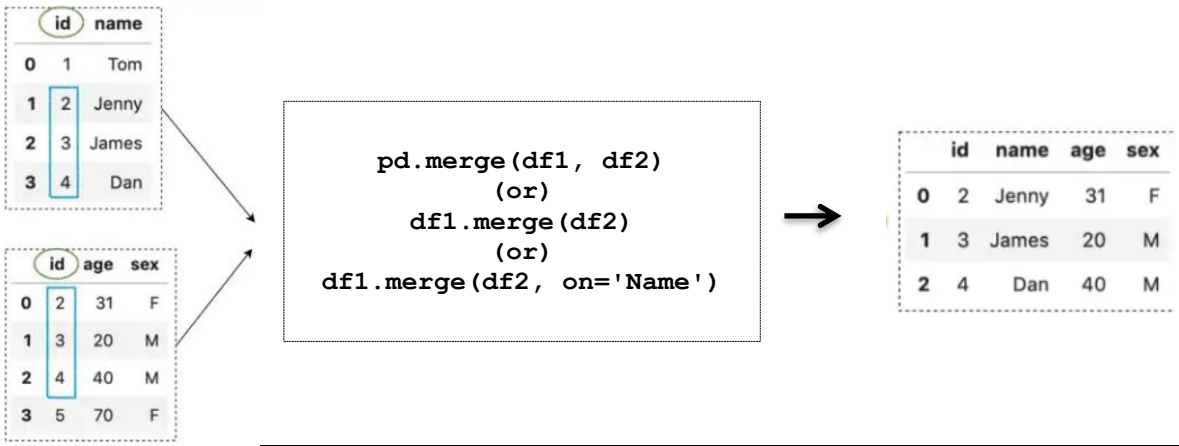
final = pd.merge(df1, df2)
print("\n",final)
```

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
```

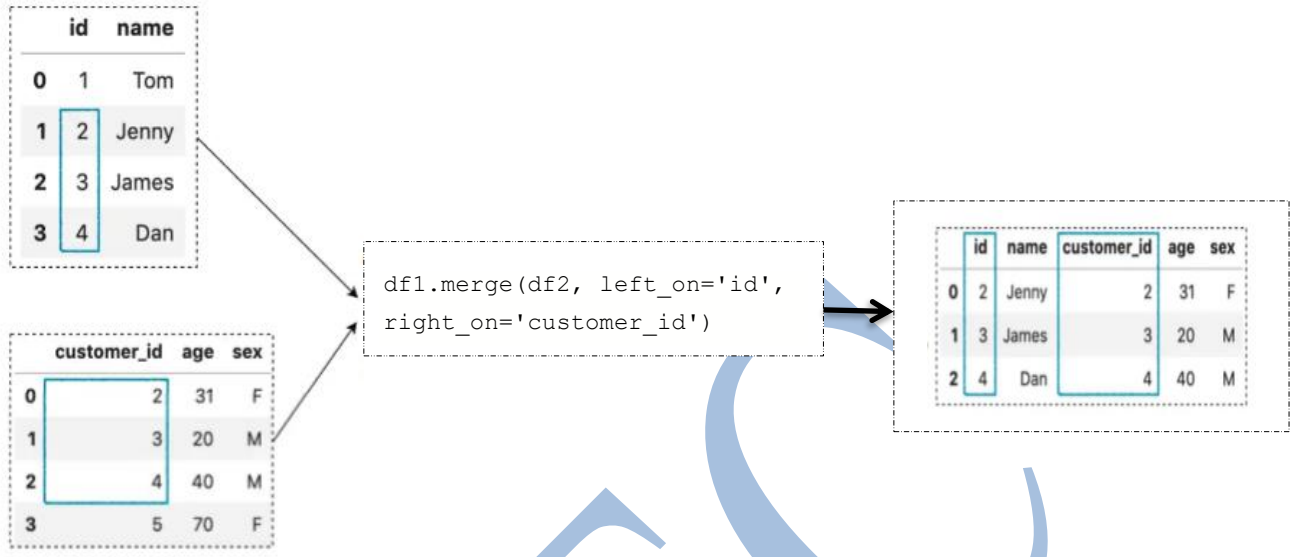


Code 2#: Merge two DataFrames via 'id' column.

Output:

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

```
   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')
```

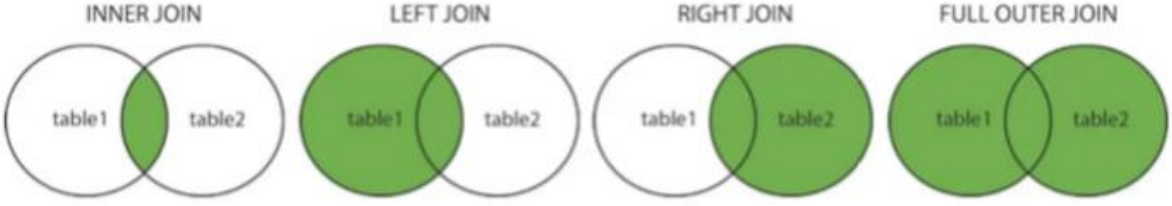
Output:

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

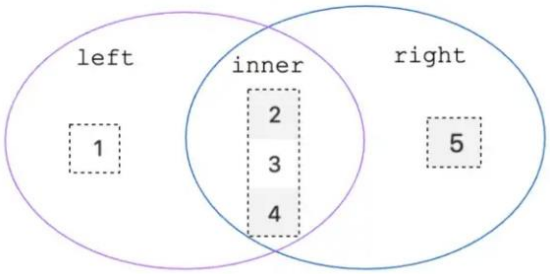
Various type of joins: inner, left, right and outer

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=?)
```



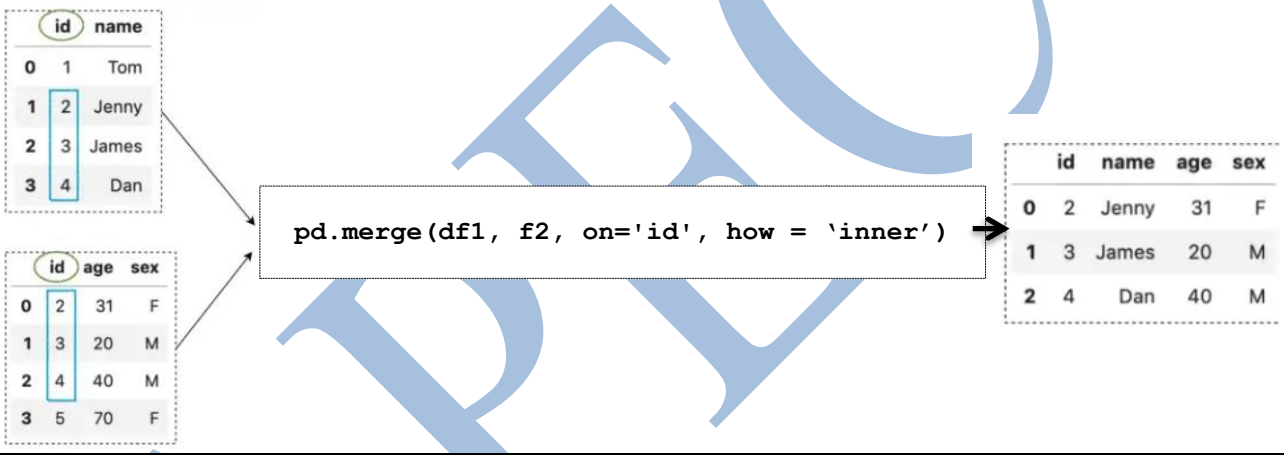
Code 4#: Merge using inner join

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
```



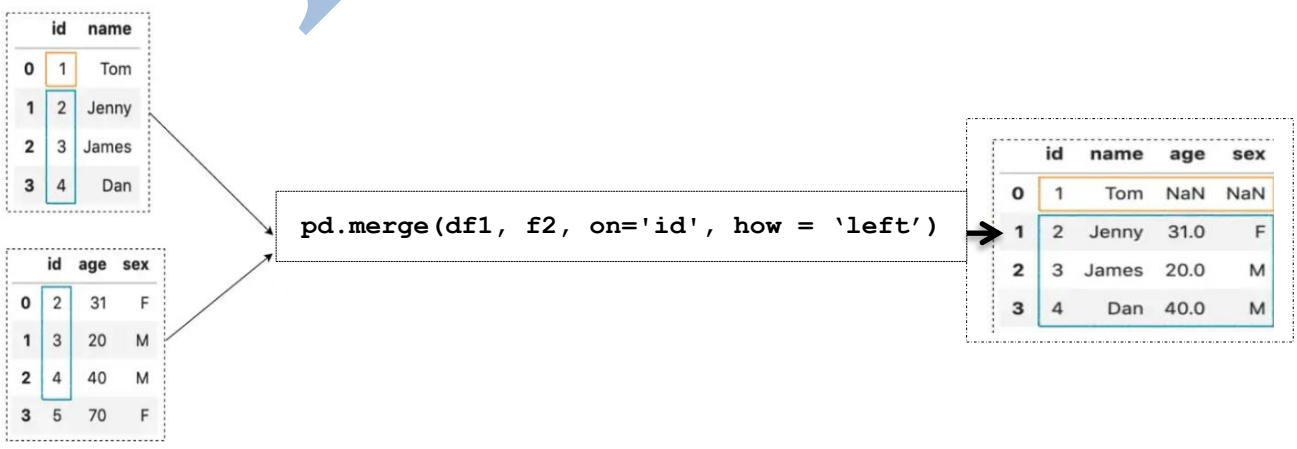
Code 4#: Merge using Left join

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
```



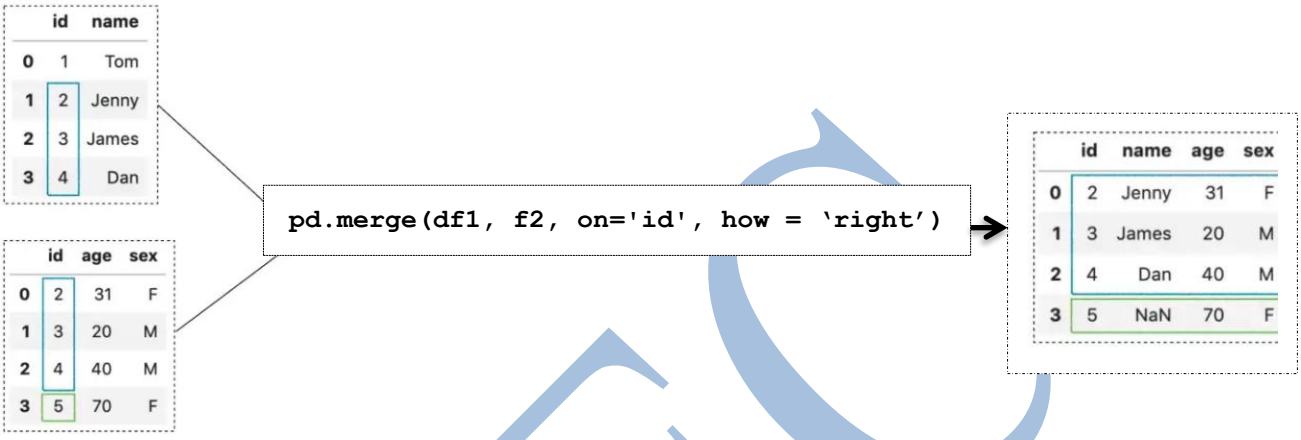
Code 4#: Merge using Right join

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
```



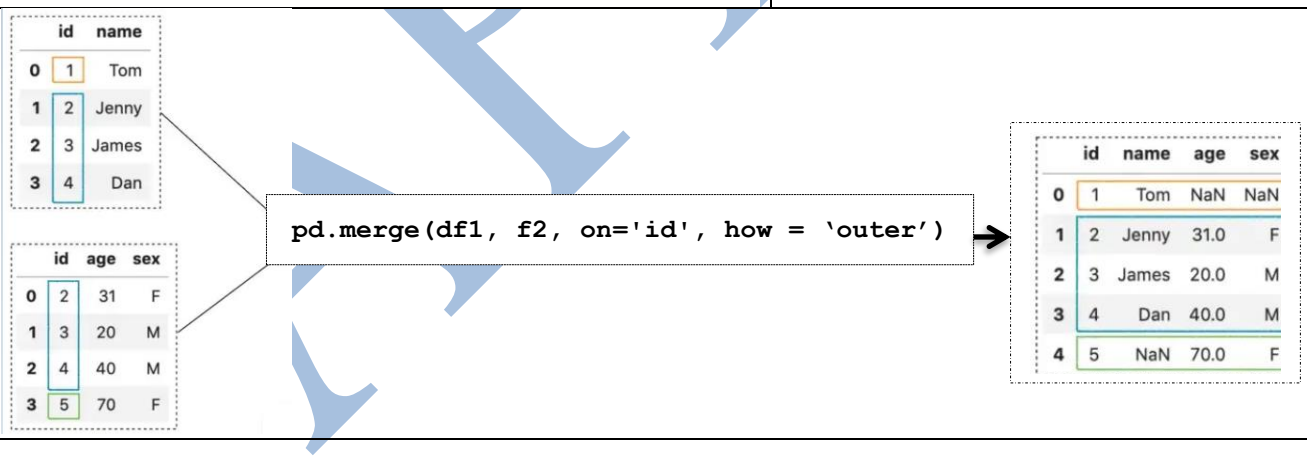
Code 4#: Merge using Outer join

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')
```

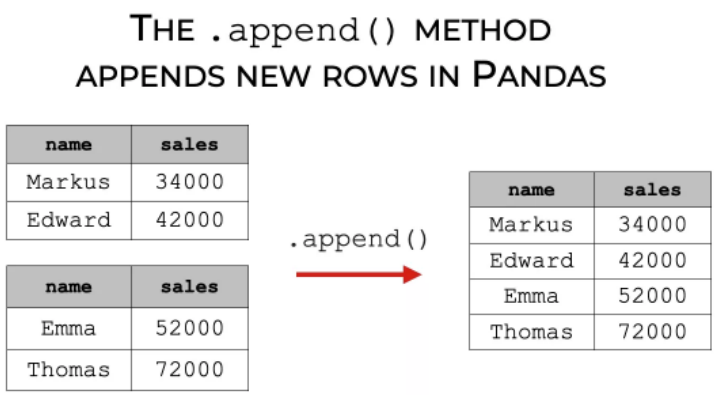
Output:

```
   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
```



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

Output:

```
   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.

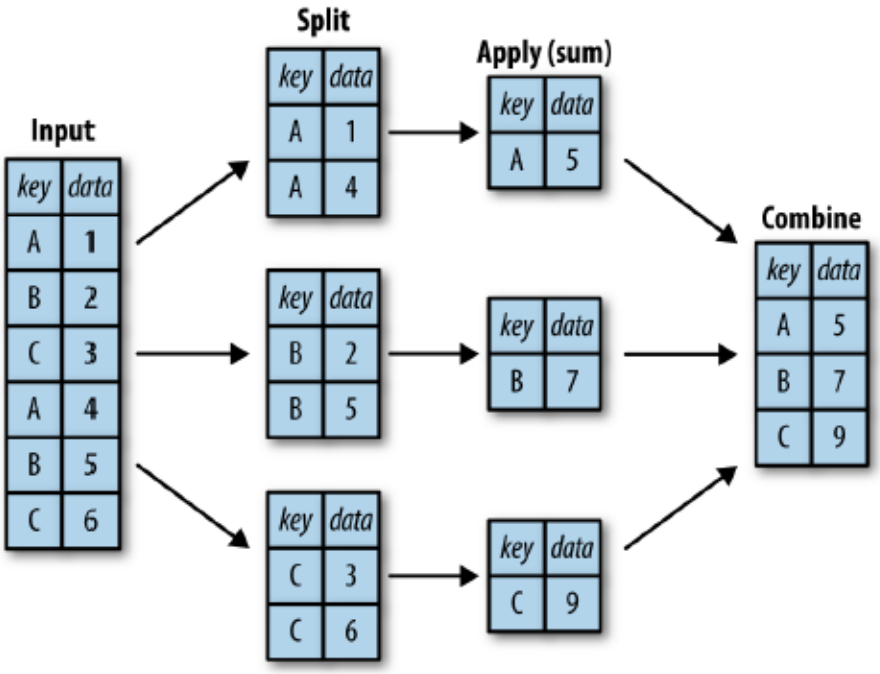
```
df.groupby(by=['Maths'])
```

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
`df.mean()`	dtype: object 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)
```

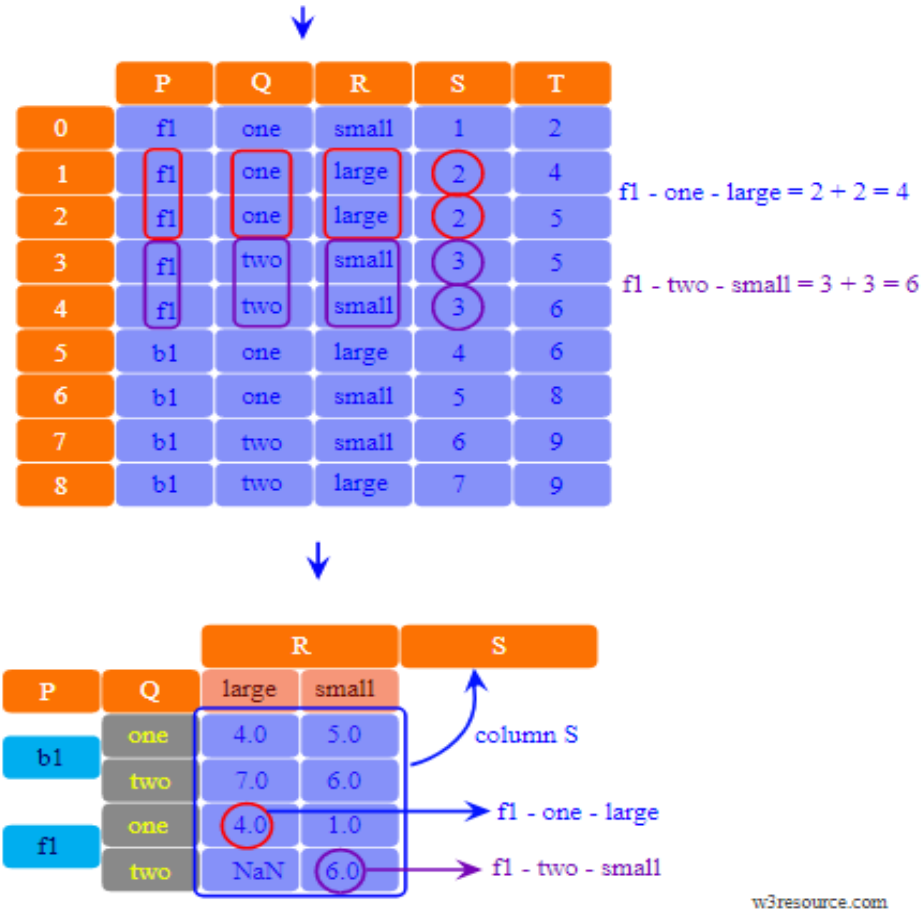
```
                     sum          mean  count
                  Salary        Salary Salary
Type
Full-time Employee  50000  16666.666667      3
Intern               5000   5000.000000      1
Part-time Employee  10000  10000.000000      1
Grand Total         65000  13000.000000      5
```

```
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

Numpy programs

Program	Output
import numpy as np a =np.array([1,2,3]) print a	[1, 2, 3]
# more than one dimensions import numpy as np a =np.array([[1,2],[3,4]]) print a	[[1, 2] [3, 4]]
# minimum dimensions import numpy as np a =np.array([1,2,3,4,5],ndmin=2) print a	[[1, 2, 3, 4, 5]]
# dtype parameter import numpy as np a =np.array([1,2,3],dtype= complex) print a	[1.+0.j, 2.+0.j, 3.+0.j]
# this resizes the ndarray import numpy as np a =np.array([[1,2,3],[4,5,6]]) **a.shape=(3,2)** print a	[[1, 2] [3, 4] [5, 6]]

import numpy as np a =np.array([[1,2,3],[4,5,6]]) **b =a.reshape(3,2)** print b	[[1, 2] [3, 4] [5, 6]]
#This array attribute returns the number of array dimensions. # an array of evenly spaced numbers import numpyas np a =np.arange(24) print a	[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
# this is one dimensional array import numpy as np a = np.arange(24) print(a) b = a.reshape(2,4,3) print(b) # b is having three dimensions	[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23] [[[0, 1, 2] [3, 4, 5] [6, 7, 8] [9, 10, 11]] [[12, 13, 14] [15, 16, 17] [18, 19, 20] [21, 22, 23]]]
# dtype of array is int8 (1 byte) importnumpyas np x =np.array([1,2,3,4,5],dtype= np.int8) printx.itemsize	1
numpy.empty It creates an uninitialized array of specified shape and dtype. It uses the following constructor **numpy.empty(shape, dtype = float, order = 'C')** import numpy as np x =np.empty([3,2],dtype=int) print x	[[22649312 1701344351] [1818321759 1885959276] [16779776 156368896]]
numpy.zeros Returns a new array of specified size, filled with zeros. **numpy.zeros(shape, dtype = float, order = 'C')** # array of five zeros. Default dtype is float import numpy as np x =np.zeros(5) print x	[0. 0. 0. 0. 0.]
import numpy as np x =np.zeros((5,),dtype= np.int) print x	[0 0 0 0 0]
# array of five ones. Default dtype is float import numpy as np x =np.ones(5) print x	[1. 1. 1. 1. 1.]

APEC

import numpy as np x =np.ones([2,2],dtype=int) print x	[[1 1] [1 1]]
# convert list to ndarray import numpy as np x =[1,2,3] a =np.asarray(x) print a	[1 2 3]
# dtype is set import numpy as np x =[1,2,3] a =np.asarray(x,dtype=float) print a	[1. 2. 3.]
# ndarray from tuple import numpy as np x =(1,2,3) a =np.asarray(x) print a	[1 2 3]
# ndarray from list of tuples import numpy as np x =[(1,2,3),(4,5)] a =np.asarray(x) print a	[(1, 2, 3) (4, 5)]
import numpy as np x =np.arange(5) print x	[0 1 2 3 4]

import numpy as np
# dtype set 	
x =np.arange(5,dtype=float)
print x
	[0. 1. 2. 3. 4.]

# start and stop parameters set import numpy as np x =np.arange(10,20,2) print x	[10 12 14 16 18]
numpy.linspace This function is similar to arange() function. In this function, instead of step size, the number of evenly spaced values between the interval is specified. import numpy as np x =np.linspace(10,20,5) print x	[10. 12.5 15. 17.5 20.]
import numpy as np arr = np.random.random(size = 2) print ("Random Array : ", arr)	1st time execution: Random Array : [0.19013078 0.71429686] 2nd time execution: Random Array : [0.34517996 0.01934729]

APEC

# Create a 3x3 array of uniformly distribute random values between 0 and 1 import numpy as np arr = np.random.random((3,3)) print ("Random Array : ", arr)	Random Array : [[0.47513993 0.76396525 0.09788919] [0.82925529 0.66002459 0.23351154] [0.58434193 0.07858471 0.57366724]]
# Create a 3x3 array of normally distributed random values # with mean 0 and standard deviation 1 import numpy as np arr = np.random.normal(0,1,(3,3)) print ("Random Array : ", arr)	Random Array : [[-2.21914362 -1.63076144 -1.06029911] [0.46289967 0.37558323 -0.43023351] [-0.74291569 2.11293305 -0.83897918]]
# Create a 3x3 array of random integers in the interval [0, 10] import numpy as np arr = np.random.randint(0,10,(3,3)) print ("Random Array : ", arr)	Random Array : [[5 0 9] [0 8 2] [2 4 4]]
#The eye() function is used to create a 2-D array with ones on the diagonal and zeros elsewhere. import numpy as np arr = np.eye(2) print (arr)	[[1. 0.] [0. 1.]]
import numpy as np arr = np.eye(3) print (arr)	[[1. 0. 0.] [0. 1. 0.] [0. 0. 1.]]
# Create an uninitialized array of three integers # The values will be whatever happens to already exist at that # memory location import numpy as np arr = np.empty(3) print (arr)	Each time you get 1 value
# endpoint set to false import numpy as np x =np.linspace(10,20,5, endpoint =False) print x	[10. 12. 14. 16. 18.]
# find retstep value import numpy as np x =np.linspace(1,2,5,retstep=True) print x # retstep here is 0.25	(array([1. , 1.25, 1.5 , 1.75, 2.]), 0.25)

APEC

NumPy Standard Data Types

Data Types in Python

strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
integer - used to represent integer numbers. e.g. -1, -2, -3
float - used to represent real numbers. e.g. 1.2, 42.42
boolean - used to represent True or False.
complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

Below is a list of all data types in NumPy and the characters used to represent them.

- ✓ i - integer
- ✓ b - boolean
- ✓ u - unsigned integer
- ✓ f - float
- ✓ c - complex float
- ✓ m - timedelta
- ✓ M - datetime
- ✓ - object
- ✓ S - string
- ✓ U - unicode string
- ✓ V - fixed chunk of memory for other type (void)
- ✓

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

The Basics of NumPy Arrays

Attributes of arrays
Determining the size, shape, memory consumption, and data types of arrays
Indexing of arrays
Getting and setting the value of individual array elements
Slicing of arrays
Getting and setting smaller subarrays within a larger array
Reshaping of arrays
Changing the shape of a given array
Joining and splitting of arrays
Combining multiple arrays into one, and splitting one array into many

APEC

NumPyufuncs:ufuncs stands for "Universal Functions"

```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []

for i, j in zip(x, y):
  z.append(i + j)
print(z)

Output:
[5, 7, 9, 11]
```

```
import numpy as np

x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = np.add(x, y)

print(z)

Output:
[5, 7, 9, 11]
```

Arithmetic Operations

Program	Output
import numpy as np arr1 = np.array([100, 11, 12, 13, 14, 15]) arr2 = np.array([20, 21, 22, 23, 24, 25]) newarr = np.add(arr1, arr2) print(newarr)	[120 32 34 36 38 40]
import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([20, 21, 22, 23, 24, 25]) newarr = np.subtract(arr1, arr2) print(newarr)	[-10 -1 8 17 26 35]
import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([20, 21, 22, 23, 24, 25]) newarr = np.multiply(arr1, arr2) print(newarr)	[200 420 660 920 1200 1500]
import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([3, 5, 10, 8, 2, 33]) newarr = np.divide(arr1, arr2) print(newarr)	[3.33333333 4. 3. 5. 25. 1.81818182]
import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([3, 5, 6, 8, 2, 33]) newarr = np.power(arr1, arr2) print(newarr) Note: 10*10*10, 20*20*20*20*20*20 etc	[1000 3200000 729000000 6553600000000 2500 0]
import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([3, 7, 9, 8, 2, 33]) newarr = np.mod(arr1, arr2) print(newarr) Note: (10%3),(20%7),(30%9) etc.	[1 6 3 0 0 27]

import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([3, 7, 9, 8, 2, 33]) newarr = np.remainder(arr1, arr2) print(newarr)	[1 6 3 0 0 27]
The divmod() function return both the quotient and the the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod. import numpy as np arr1 = np.array([10, 20, 30, 40, 50, 60]) arr2 = np.array([3, 7, 9, 8, 2, 33]) newarr = np.divmod(arr1, arr2) print(newarr)	(array([3, 2, 3, 5, 25, 1]), array([1, 6, 3, 0, 0, 27]))
import numpy as np arr = np.array([-1, -2, 1, 2, 3, -4]) newarr = np.absolute(arr) print(newarr)	[1 2 1 2 3 4]
import numpy as np array1 = np.array([[10, 20, 30], [40, 50, 60]]) print("Mean: ", np.mean(array1)) print("Std: ", np.std(array1)) print("Var: ", np.var(array1)) print("Sum: ", np.sum(array1)) print("Prod: ", np.prod(array1))	Mean: 35.0 Std: 17.07825127659933 Var: 291.6666666666667 Sum: 210 Prod: 720000000
import numpy as np arr = np.array([10, 20, 30]) print("Max Element: ", np.max(arr)) print("Max Element is at loaction : ", np.argmax(arr)) print("Min Element: ", np.min(arr)) print("Mix Element is at loaction : ", np.argmin(arr))	Max Element: 30 Max Element is at loaction : 2 Min Element: 10 Mix Element is at loaction : 0
import numpy as np arr = np.array([[10, 2], [3, 48]]) print(arr) print("Max Element: ", np.max(arr)) print("Max Element is at loaction : ", np.argmax(arr))	[[10 2] [3 48]] Max Element: 48 Max Element is at loaction : 3
import numpy as np arr = np.array([[10, 2], [3, 48]]) print(arr) print("Max Element: ", np.max(arr)) print("Max Element is at location :", np.argmax(arr,axis = 0))	[[10 2] [3 48]] Max Element: 48 Max Element is at location : [0 1]
import numpy as np arr = np.array([[[10, 2, 7], [334, 48,9], [4,7,23]]]) print(arr) print("Max Element: ", np.max(arr)) print("Max Element is at loaction : ", np.argmax(arr))	[[[10 2 7] [334 48 9] [4 7 23]]] Max Element: 334 Max Element is at location : 3
What is the difference between summation and addition? Addition is done between two arguments whereas summation happens over n elements.	
import numpy as np arr1 = np.array([1, 2, 3]) arr2 = np.array([1, 2, 3]) newarr = np.add(arr1, arr2) print(newarr) Output [2 4 6]	import numpy as np arr1 = np.array([1, 2, 3]) arr2 = np.array([1, 2, 3]) newarr = np.sum([arr1, arr2]) print(newarr) Output 12

Rounding Decimals

There are primarily five ways of rounding off decimals in NumPy:

- truncation
- fix
- rounding
- floor
- ceil

#Remove the decimals, and return the float number closest to zero. import numpy as np arr = np.trunc([-3.1666, 3.6667]) print(arr)	[-3. 3.]
#Remove the decimals, and return the float number closest to zero. import numpy as np arr = np.fix([-3.1666, 3.6667]) print(arr)	[-3. 3.]
Rounding The around() function increments preceding digit or decimal by 1 if >=5 else do nothing. E.g. round off to 1 decimal point, 3.16666 is 3.2 import numpy as np arr = np.around(3.1666, 2) print(arr)	3.17
import numpy as np arr = np.floor([-3.1666, 3.6667]) print(arr)	[-4. 3.]
import numpy as np arr = np.ceil([-3.1666, 3.6667]) print(arr)	[-3. 4.]

APEC

Logs

NumPy provides functions to perform log at the base 2, e and 10.

#Log at Base 2 #Use the log2() function to perform log at the base 2. import numpy as np arr = np.arange(1, 10) print(np.log2(arr))	[0. 1. 1.5849625 2. 2.32192809 2.5849625 2.80735492 3. 3.169925]
import numpy as np arr = np.arange(1, 10) print(np.log10(arr))	[0. 0.30103 0.47712125 0.60205999 0.69897 0.77815125 0.84509804 0.90308999 0.95424251]
#log() function to perform log at the base e. import numpy as np arr = np.arange(1, 10) print(np.log(arr))	[0. 0.69314718 1.09861229 1.38629436 1.60943791 1.79175947 1.94591015 2.07944154 2.19722458]

Summations

#Summation Over an Axis #If you specify axis=1, NumPy will sum the numbers in each array. import numpy as np arr1 = np.array([13, 2, 3]) arr2 = np.array([1, 2, 3]) newarr = np.sum([arr1, arr2], axis=1) print(newarr)	[18 6]
#Cummulative sum means partially adding the elements in array. import numpy as np arr = np.array([1, 2, 3, 4, 5, 6]) newarr = np.cumsum(arr) print(newarr)	[1 3 6 10 15 21] *#E.g. The partial sum of [1, 2, 3, 4] would be [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10].*

NumPy Products

- To find the product of the elements in an array, use the prod() function.

import numpy as np arr = np.array([1, 2, 3, 4]) x = np.prod(arr) print(x)	24 *Returns: 24 because 1*2*3*4 = 24*
import numpy as np arr1 = np.array([1, 2, 3, 4]) arr2 = np.array([5, 6, 7, 8]) x = np.prod([arr1, arr2]) print(x)	40320 *Returns: 40320 because 1*2*3*4*5*6*7*8 = 40320*

#Product Over an Axis import numpy as np arr1 = np.array([1, 2, 3, 4]) arr2 = np.array([5, 6, 7, 8]) newarr = np.prod([arr1, arr2], axis=1) print(newarr)	[24 1680] *Because* *1*2*3*4 = 24* *5*6*7*8 = 1680*
#Cummulative product means taking the product partially. import numpy as np arr = np.array([5, 6, 7, 8]) newarr = np.cumprod(arr) print(newarr)	[5 30 210 1680] *Because 5 5*6 5*6*7 5*6*7*8*

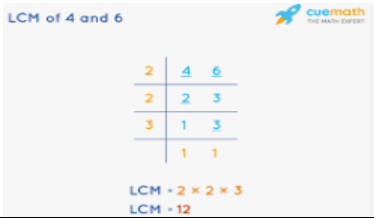
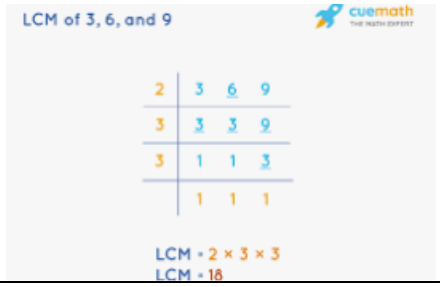
NumPy Differences

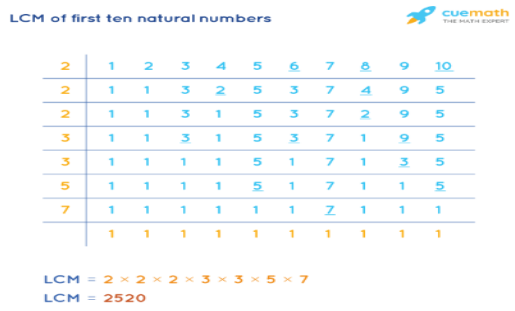
- A discrete difference means subtracting two successive elements.
- E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]
- To find the discrete difference, use the diff() function.

import numpy as np arr = np.array([10, 15, 25, 5]) newarr = np.diff(arr) print(newarr)	[5 10 -20] *Because 15-10 25-15 5-25*
#Compute discrete difference of the following array twice: import numpy as np arr = np.array([10, 15, 25, 5]) newarr = np.diff(arr, n=2) print(newarr)	[5 -30] *Because 15-10=5, 25-15=10, and 5-25=-20* *AND* *10-5=5 and -20-10=-30*

NumPy LCM Lowest Common Multiple

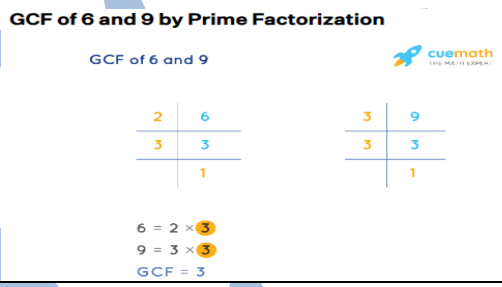
In Mathematics, the LCM of any two is the value that is evenly divisible by the two given numbers.

import numpy as np num1 = 4 num2 = 6 x = np.lcm(num1, num2) print(x)	12 
#Finding LCM in Arrays #To find the Lowest Common Multiple of all values in an array, use the reduce() method. import numpy as np arr = np.array([3, 6, 9]) x = np.lcm.reduce(arr) print(x)	18 
import numpy as np	[1 2 3 4 5 6 7 8 9 10]

arr = np.arange(1, 11) print(arr) x = np.lcm.reduce(arr) print(x)	2520 

NumPy GCD Greatest Common Denominator

- The GCD (Greatest Common Denominator), also known as HCF (Highest Common Factor) is the biggest number that is a common factor of both of the numbers.

import numpy as np num1 = 6 num2 = 9 x = np.gcd(num1, num2) print(x)	3 
import numpy as np arr = np.array([20, 8, 32, 36, 16]) x = np.gcd.reduce(arr) print(x)	4

NumPy Trigonometric Functions

- NumPy provides the ufuncs sin(), cos() and tan() that take values in radians and produce the corresponding sin, cos and tan values.

#Find sine value of PI/2: import numpy as np x = np.sin(np.pi/2) print(x)	1.0
import numpy as np arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5]) x = np.sin(arr) print(x)	[1. 0.8660254 0.70710678 0.58778525]
#Convert Degrees Into Radians #radians values = pi/180 * degree_values. import numpy as np arr = np.array([90, 180, 270, 360]) x = np.deg2rad(arr) print(x)	[1.57079633 3.14159265 4.71238898 6.28318531]
#Radians to Degrees import numpy as np arr = np.array([np.pi/2, np.pi, 1.5*np.pi, 2*np.pi]) x = np.rad2deg(arr) print(x)	[90. 180. 270. 360.]

Finding Angles Finding angles from values of sine, cos, tan. E.g. sin, cos and tan inverse (arcsin, arccos, arctan). NumPy provides ufuncsarcsin(), arccos() and arctan() that produce radian values for corresponding sin, cos and tan values given. import numpy as np x = np.arcsin(1.0) print(x)	1.5707963267948966
#Angles of Each Value in Arrays import numpy as np arr = np.array([1, -1, 0.1]) x = np.arcsin(arr) print(x)	[1.57079633 -1.57079633 0.10016742]

NumPy Set Operations

- A set in mathematics is a collection of unique elements.
- Sets are used for operations involving frequent intersection, union and difference operations.

import numpy as np arr = np.array([1, 1, 1, 2, 3, 4, 5, 5, 6, 7]) x = np.unique(arr) print(x)	[1 2 3 4 5 6 7]
import numpy as np arr1 = np.array([1, 2, 3, 4]) arr2 = np.array([3, 4, 5, 6]) newarr = np.union1d(arr1, arr2) print(newarr)	[1 2 3 4 5 6]
import numpy as np arr1 = np.array([1, 2, 3, 4]) arr2 = np.array([3, 4, 5, 6]) newarr = np.intersect1d(arr1, arr2, assume_unique=True) print(newarr)	[3 4]
#To find only the values in the first set that is NOT present in the second import numpy as np set1 = np.array([1, 2, 3, 4]) set2 = np.array([3, 4, 5, 6]) newarr = np.setdiff1d(set1, set2, assume_unique=True) print(newarr)	[1 2]
#Find only the values that are NOT present in BOTH sets import numpy as np	[1 2 5 6]

set1 = np.array([1, 2, 3, 4]) set2 = np.array([3, 4, 5, 6]) newarr = np.setxor1d(set1, set2, assume_unique=True) print(newarr)	

APEC

Comparisons, Masks, and Boolean Logic

Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Numpy: Boolean Indexing

<table>
<tr><td>import numpy as np
A = np.array([4, 7, 3, 4, 2, 8])
print(A == 4)</td><td>
[True False False True False False]</td></tr>
<tr><td colspan="2">Every element of the Array A is tested, if it is equal to 4. The results of these tests are the Boolean elements of the result array.
Of course, it is also possible to check on "<", "<=", ">" and ">=".</td></tr>
<tr><td>import numpy as np
A = np.array([4, 7, 3, 4, 2, 8])
print(A < 5)

<u>Output:</u>
[True False True True True False]</td><td>import numpy as np
B = np.array([[42,56,89,65],
[99,88,42,12],
[55,42,17,18]])

print(B>=42)

<u>Output:</u>
[[True True True True]
[True True True False]
[True True False False]]</td></tr>
<tr><td>import numpy as np
A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9],
[11, 14, 12, 15, 15, 16, 10, 12, 11],
[10, 12, 12, 15, 14, 16, 10, 12, 12]])

print(A < 15)

<u>Output:</u>
[[True True True True False True True True True]
[True True True False False False True True True]
[True True True False True False True True True]]</td><td>import numpy as np
A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9],
[11, 14, 12, 15, 15, 16, 10, 12, 11],
[10, 12, 12, 15, 14, 16, 10, 12, 12]])
B = A < 15
print(B.astype(np.int32))

<u>Output:</u>
[[1 1 1 1 0 1 1 1 1]
[1 1 1 0 0 0 1 1 1]
[1 1 1 0 1 0 1 1 1]]</td></tr>
<tr><td></td><td></td></tr>
</table>

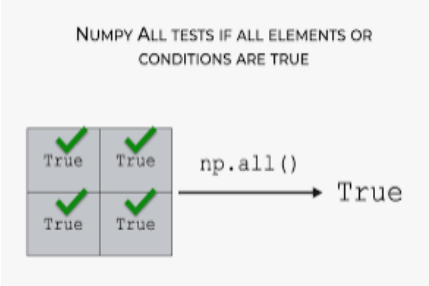
APEC

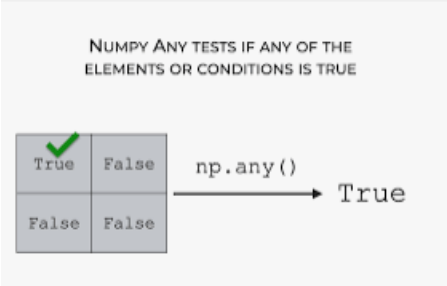
Comparison operators

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Working with Boolean Arrays

np.count_nonzero(A < 13) = np.sum(A < 13)	
import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) **B = np.count_nonzero(A < 13)** print("Values less than 13: ", B) *Output:* Values less than 13: 10	import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) **B = np.sum(A < 13)** print("Values less than 13: ", B) *Output:* Values less than 13: 10
import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) B = np.sum(A < 13, axis=1) print("Values less than 13: ", B) *Output:* Values less than 13: [5 5]	





#Are all values greater than 8 import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) print(np.all(A > 8)) *Output:* True	# Are there any values less than zero? import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) print(np.any(A == 8)) *Output:* False

The NumPy library in Python is a popular library for working with arrays. Boolean masking, also called boolean indexing, is a feature in Python NumPy that allows for the filtering of values in numpy arrays.

There are two main ways to carry out boolean masking:

Method one: Returning the result array.
Method two: Returning a boolean array.

Boolean operators

Operator	Equivalent ufunc
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) **B = np.sum((A < 13) & (A > 11))** print("Values between 11 & 13: ", B)	Values between 11 & 13: 4
import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) **B = np.sum((A != 9))** print("Values not equal to 9: ", B)	Values not equal to 9: 17
Boolean-valued indexing: NumPy also permits the use of a boolean-valued array as an index, to perform advanced indexing on an array. I	
import numpy as np A = np.array([[12, 13, 14, 12, 16, 14, 11, 10, 9], [11, 14, 12, 15, 15, 16, 10, 12, 11]]) arr = np.sum(A) print("Sum of the array : ", arr) B = np.sum((A < 11)) print("No. of Values less than 11 : ", B) C = A[A < 11] print("List of Values less than 11 : ", C) D = np.sum(A[A < 11]) print("Sum of Values less than 11 : ", D)	Values less than 11 : 3 Sum of Values less than 11 : 29
import numpy as np C = np.array([123,188,190,99,77,88,100]) A = np.array([4,7,2,8,6,9,5]) **R = C[A<=5]** print(R)	[123 190 100]
import numpy as np A = np.array([1,2,3,4,5,10,12,14,23,45,76,99]) div3 = A[A%3!=0] print("Array Elements not divisible by 3:") print(div3) print("----------") div5 = A[A%5==0] print("Array Elements A divisible by 5:") print(div5)	Array Elements not divisible by 3: [1 2 4 5 10 14 23 76] ---------- Array Elements A divisible by 5: [5 10 45] ---------- Array Elements, which are divisible by 3 and 5: [45] ---------- Array Elements divisible by 3: [3 12 45 99] ---------- Original Array Elements: [1 2 3 4 5 10 12 14 23 45 76 99]

print("------------------")
 	
print("Array Elements, which are divisible by 3 and 5:")
print(A[(A%3==0) & (A%5==0)])
print("------------------")
 	
div3 = A[A%3==0]
print("Array Elements divisible by 3:")
print(div3)
print("------------------")
 	
#Assign the value 1000 to elements which are divisible by 3
print("Original Array Elements: ", A)
#Modifying Values with Fancy Indexing
A[A%3==0] = 1000
print("""Assign the value of 1000 to elements which are divisible by 3:""")
print(A)
	Assign the value of 1000 to elements which are divisible by 3:
	[1 2 1000 4 5 10 1000 14 23 1000 76 1000]

A masked array is the combination of a standard numpy. ndarray and a mask. A mask is either nomask , indicating that no value of the associated array is invalid, or an array of booleans that determines for each element of the associated array whether the value is valid or not.

import numpy as np
A = np.arange(11, 99, 11)
print("Original array :", A)
indices = [2, 4, 6]
B = A[indices]
print("Partial array :", B)
 	
B[0] = -1000 # this does not affect A
print("Changing one element in the Partial array :", B)
 	
A[indices] = -1 # this alters A
print("Changing the Original array :", A)
 	
Output
Original array : [11 22 33 44 55 66 77 88]
Partial array : [33 55 77]
Changing one element in the Partial array : [-1000 55 77]
Changing the Original array : [11 22 -1 44 -1 66 -1 88]	

# importing NumPy 	
import numpy as np
# Creating a NumPy array 	
arr = np.array([[0, 9, 0],
[0, 7, 8],
[6, 0, 1]])
# Printing our array to observe 	
print(arr)
# Creating a mask array 	
mask = arr > 5
# Printing the mask array 	
print(mask)
# Printing the filtered array using both methods 	
	[[0 9 0]
	[0 7 8]
	[6 0 1]]
	[[False True False]
	[False True True]
	[True False False]]
	[9 7 8 6]
	[9 7 8 6]

print(arr[mask]) print(arr[arr > 5])	

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once.

import numpy as np a = np.arange(10,100, 10) print("Given array : ", a) indices = np.array([2, 3, 4]) print("Array values at locations [2,3,4] : ", a[indices]) *Output* Given array : [10 20 30 40 50 60 70 80 90] Array values at locations [2,3,4] : [30 40 50]	import numpy as np a = np.linspace(1,100, 5) print("Given array : ", a) indices = np.array([2, 3, 4]) print("Array values at locations [2,3,4] : ", a[indices]) *Output* Given array : [1. 25.75 50.5 75.25 100.] Array values at locations [2,3,4] : [50.5 75.25 100.]
Combined Indexing For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen: In[9]: print(X) [[0 1 2 3] [4 5 6 7] [8 9 10 11]] **We can combine fancy and simple indices:** In[10]: X[2, [2, 0, 1]] Out[10]: array([10, 8, 9]) **We can also combine fancy indexing with slicing:** In[11]: X[1:, [2, 0, 1]] Out[11]: array([[6, 4, 5], [10, 8, 9]]) **And we can combine fancy indexing with masking:** In[12]: mask = np.array([1, 0, 1, 0], dtype=bool) X[row[:, np.newaxis], mask] Out[12]: array([[0, 2], [4, 6], [8, 10]]) All of these indexing options combined lead to a	

structured arrays

A structure array is a data type that groups related data using data containers called fields. Each field can contain any type of data. Access data in a field using dot notation of the form structName.

A structure array is a data type that groups related data using data containers called fields. Each field can contain any type of data. Access data in a field using dot notation of the form structName.fieldName.