UNIT VI: Pandas

Python Pandas Module

1. A pandas is an **open source library** in Python. It provides ready to use high-performance data structures and data analysis tools.
2. Pandas module runs on top **of NumPy** and it is popularly used for data science and data analytics.
3. NumPy is a low-level data structure that supports multi-dimensional arrays and a wide range of mathematical array operations. Pandas have a higher-level interface. It also provides streamlined alignment of tabular data and powerful time series functionality.
4. DataFrame is the key data structure in Pandas. It allows us to store and manipulate tabular data as a 2-D data structure.
5. Pandas provide a rich feature-set on the DataFrame. For example, data alignment, data statistics, slicing, grouping, merging, concatenating data, etc**.**

Installing and Getting Started with Pandas

You need to have Python 2.7 and above to install Pandas module. If you are using **conda**, then you can install it using below command.

```
conda install pandas
```

If you are using PIP, then run the below command to install pandas module.

```
pip3.7 install pandas
```

To import Pandas and NumPy in your Python script, add the below piece of code:

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

As Pandas is dependent on the NumPy library, we need to import this dependency.

Data Structures in Pandas module

There are **3 data structures** provided by the **Pandas module**, which are as follows:

- **Series**: It is a **1-D size-immutable array like structure** having homogeneous data.
- **DataFrames**: It is a **2-D size-mutable tabular structure with heterogeneously typed columns**.
- **Panel**: It is a **3-D, size-mutable array.**

What Can You Do With DataFrames Using Pandas?

Pandas make it simple to do many of the time consuming, repetitive tasks associated with working with data, including:

- Data cleansing
- Data fill
- Data normalization
- Merges and joins
- Data visualization
- Statistical analysis
- Data inspection
- Loading and saving data

- And much more

Pandas DataFrame

DataFrame is the most important and widely used data structure and is a standard way to store data. DataFrame has data aligned in **rows and columns like the SQL table or a spreadsheet database**. We can either hard code data into a DataFrame or import a CSV file, tsv file, Excel file, SQL table, etc. We can use the below constructor for creating a DataFrame object.

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

Below is a short description of the parameters:

- **data** - create a **DataFrame object from the input data**. It can be list, dict, series, Numpy ndarrays or even, any other DataFrame.
- **index** - has the **row labels**
- **columns** - used to create **column labels**
- **dtype** - used to specify the **data type of each column, optional parameter**
- **copy** - used for **copying data, if any**

There are many ways to create a DataFrame. We can create DataFrame object from Dictionaries or list of dictionaries. We can also create it from a list of tuples, CSV, Excel file, etc. Let's run a simple code to create a DataFrame from the **list of dictionaries**.

```
import pandas as pd
import numpy as np
df = pd.DataFrame({
   "State": ['Andhra Pradesh', 'Maharashtra', 'Karnataka', 'Kerala', 'Tamil Nadu'],
   "Capital": ['Hyderabad', 'Mumbai', 'Bengaluru', 'Trivandrum', 'Chennai'],
   "Literacy %": [89, 77, 82, 97,85],
   "Avg High Temp(c)": [33, 30, 29, 31, 32 ]
})
print(df)
```

1. The first step is to create a dictionary.
2. The second step is to pass the dictionary as an argument in the DataFrame() method.
3. The final step is to print the DataFrame.
4. The DataFrame can be compared to a **table having heterogeneous value**.
5. Also, the size of the DataFrame can be modified.
6. We have supplied the data in the form of the **map and the keys of the map are considered by Pandas as the row labels**.
7. The index is displayed in the leftmost column and has the row labels.
8. The column header and data are displayed in a tabular fashion.

9. It is also possible to create indexed DataFrames. This can be done by configuring the index parameter in the **DataFrame()** method.

2. Importing data from CSV to DataFrame

1. We can also create a **DataFrame by importing a CSV file**.
2. A CSV file is a **text file with one record of data per line**.
3. The values within the record are separated using the “comma” character.
4. Pandas provides a useful method, named **read_csv()** to read the contents of the CSV file into a DataFrame.
5. For example, we can create a file named **‘cities.csv’** containing details of Indian cities. The CSV file is stored in the same directory that contains Python scripts. This file can be imported using:

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

3. Inspecting data in DataFrame

1. Running the DataFrame using its name displays the entire table.
2. In real-time, the datasets **to analyze will have thousands of rows.** .
3. For analyzing data, we need to **inspect data from huge volumes of datasets.**
4. Pandas provide many **useful functions to inspect only the data** we need.
    a. **df.head(n) to get the first n rows**
    b. **df.tail(n) to print the last n rows**. For example, the below code prints the first 2 rows and last 1 row from the DataFrame.

print(df.head(2))
print(df.tail(1))

   c. **print(df.dtypes)** prints the data types
   d. **print(df.index)** prints index
   e. **print(df.columns)** prints the columns of the DataFrame.
   f. **print(df.values)** displays the table values.

4. Getting Statistical summary of records

We can get statistical summary (**count, mean, standard deviation, min, max etc**.) of the data using **df.describe() function.**

print(df['Literacy %'].describe())

The df.describe() function displays the statistical summary, along with the data type.

5. Sorting records

We can sort records by any column using df.sort_values() function. For example, let’s sort the “Literacy %” column in descending order.

print(df.sort_values('Literacy %', ascending=False))

3. Slicing records

It is possible to **extract data of a particular column**, by **using the column name**. For example, to extract the 'Capital' column, we use:

df['Capital'] or (df.Capital)

It is also possible to slice multiple columns. This is done by enclosing multiple column names enclosed in **2 square brackets**, with the column names separated using commas. The following code slices the '**State' and 'Capital'** columns of the DataFrame.

print(df[['State', 'Capital']])

It is also possible to slice rows**. Multiple rows can be selected using ":" operator**. The below code returns the first 3 rows.

df[0:3]

An interesting feature of Pandas library is to select data based on its **row and column labels using iloc[0] function**. Many times, we may need only **few columns to analyze**. We can also can select by index using **loc['index_one']).**

For example,

to **select the second row**, we can use **df.iloc[1,:]** .

to select second element of the second column. This can be done by using **df.iloc[1,1]** function. In this example, the function **df.iloc[1,1] displays "Mumbai" as output.**

6. Filtering data

It is also possible to **filter on column values**. For example, below code filters the **columns having Literacy% above 90%.**

print(df[df['Literacy %']>90])

Any comparison operator can be used to filter, based on a condition.

Another way to filter data is using the **isin.**

Following is the code to filter only 2 states 'Karnataka' and 'Tamil Nadu'.

print(df[df['State'].isin(['Karnataka', 'Tamil Nadu'])])

5. Rename column

It is possible to use the **df.rename()** function to rename a column.

It has two arguments namely **old column name and new column name.**

For example, let's rename the column **'Literacy %**' to '**Literacy percentage'**.

df.rename(columns = {'Literacy %':'Literacy percentage'}, inplace=True)

print(df.head())

The argument `**inplace=True**` makes the changes to the **DataFrame.**

6. Data Wrangling

Data Science involves the processing of data so that the data can work well with the **data algorithms**.

Data Wrangling is the process of **processing data, like merging, grouping and concatenating**. The **Pandas library** provides useful functions like **merge(), groupby() and concat()** to support Data Wrangling tasks. Let's create **2 DataFrames** and show the Data Wrangling functions to understand it better.

```
import pandas as pd
d = {
   'Employee_id': ['1', '2', '3', '4', '5'],
   'Employee_name': ['Akshar', 'Jones', 'Kate', 'Mike', 'Tina']
}
df1 = pd.DataFrame(d, columns=['Employee_id', 'Employee_name'])
print(df1)
```

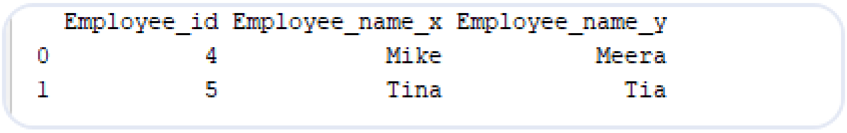
Let's create the second DataFrame using the below code:

```
import pandas as pd
data = {
   'Employee_id': ['4', '5', '6', '7', '8'],
   'Employee_name': ['Meera', 'Tia', 'Varsha', 'Williams', 'Ziva']
}
df2 = pd.DataFrame(data, columns=['Employee_id', 'Employee_name'])
print(df2)
```

a. Merging

merge the **2 DataFrames** we created, along the values of '**Employee_id'** using the merge() function:

```
print(pd.merge(df1, df2, on='Employee_id'))
```


```
  Employee_id Employee_name_x Employee_name_y
0           4            Mike           Meera
1           5            Tina             Tia
```

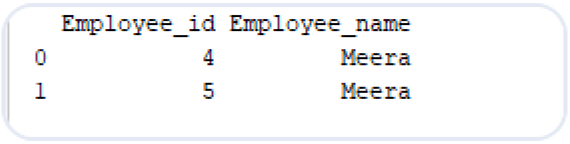
We can see that **merge() function** returns the rows from both the DataFrames having the same column value, that was used while merging.

b. Grouping

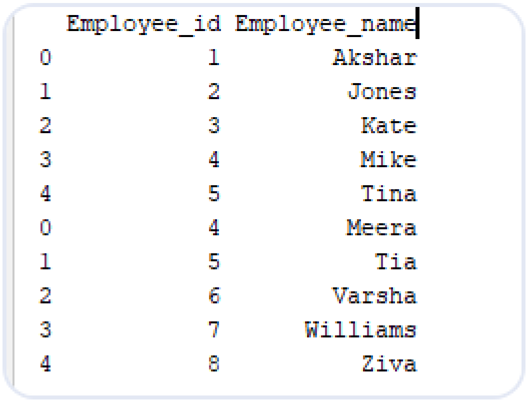
Grouping is a process of collecting data into different categories. For example, in the below example, the "Employee_Name" field has the name "Meera" two times. So, let's group it by "Employee_name" column.

```
import pandas as pd
import numpy as np
data = {
   'Employee_id': ['4', '5', '6', '7', '8'],
   'Employee_name': ['Meera', 'Meera', 'Varsha', 'Williams', 'Ziva']
}
df2 = pd.DataFrame(data)
group = df2.groupby('Employee_name')
print(group.get_group('Meera'))
```

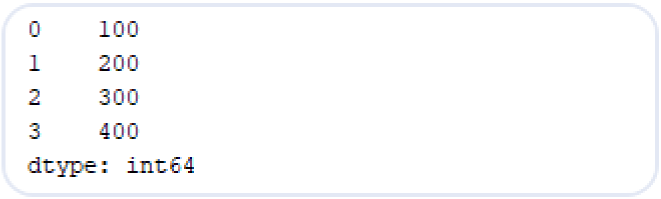
The 'Employee_name' field having value 'Meera' is grouped by the column "Employee_name".


```
  Employee_id Employee_name
0           4         Meera
1           5         Meera
```

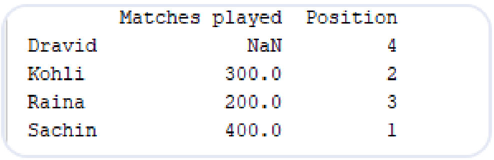
c. Concatenating: Concatenating data involves **adding one set of data to other**. Pandas provides a function named **concat()** to concatenate DataFrames. For example, let's concatenate the DataFrames df1 and df2 , using :
print(pd.concat([df1, df2]))
Output:


```
  Employee_id Employee_name
0           1        Akshar
1           2         Jones
2           3          Kate
3           4          Mike
4           5          Tina
0           4         Meera
1           5           Tia
2           6        Varsha
3           7      Williams
4           8          Ziva
```

7. Create a DataFrame by passing Dict of Series
To create a Series, we can use the **pd.Series() method** and pass an array to it. Let's create a simple Series as follows:
series_sample = pd.Series([100, 200, 300, 400])
print(series_sample)
Output:


```
0    100
1    200
2    300
3    400
dtype: int64
```

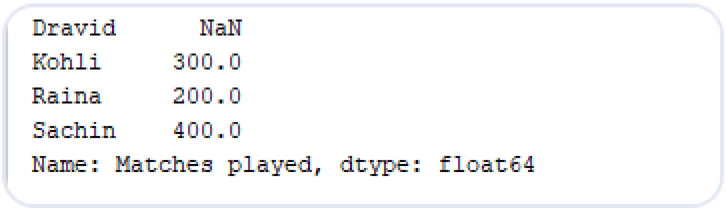
In the above, The first column contains the **index values starting from 0**. The second column contains the **elements passed as series**. It is possible to create a **DataFrame** by passing a **dictionary of `Series`**. Let's create a DataFrame that is formed by uniting and passing the indexes of the series. Example
d = {'Matches played' : pd.Series([400, 300, 200], index=['Sachin', 'Kohli', 'Raina']),
'Position' : pd.Series([1, 2, 3, 4], index=['Sachin', 'Kohli', 'Raina', 'Dravid'])}
df = pd.DataFrame(d)
print(df)


```
        Matches played  Position
Dravid             NaN         4
Kohli            300.0         2
Raina            200.0         3
Sachin           400.0         1
```

For series one, as we have not specified label 'd'**, NaN is returned.**

8. Column Selection, Addition, Deletion
It is possible to select a **specific column from the DataFrame**.
For example, to **display only the first column**, we can re-write the above code as:

d = {'Matches played' : pd.Series([400, 300, 200], index=['Sachin', 'Kohli', 'Raina']),
'Position' : pd.Series([1, 2, 3, 4], index=['Sachin', 'Kohli', 'Raina', 'Dravid'])}
df = pd.DataFrame(d)
print(df['Matches played'])
output:



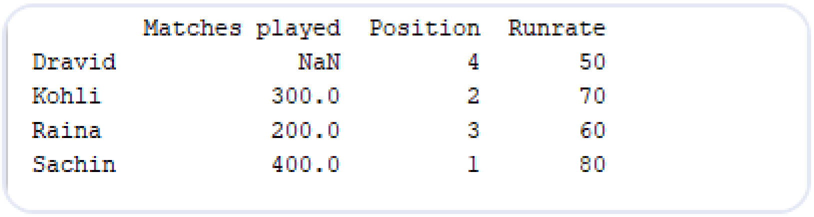
```
Dravid      NaN
Kohli     300.0
Raina     200.0
Sachin    400.0
Name: Matches played, dtype: float64
```

The above code prints only the “Matches played” column of the DataFrame.

It is also possible to **add columns to an existing DataFrame**. For example, the below code adds a **new column named “Runrate” to the above DataFrame**.

d = {'Matches played' : pd.Series([400, 300, 200], index=['Sachin', 'Kohli', 'Raina']),
'Position' : pd.Series([1, 2, 3, 4], index=['Sachin', 'Kohli', 'Raina', 'Dravid'])}
df = pd.DataFrame(d)
df['Runrate']=pd.Series([80, 70, 60, 50], index=['Sachin', 'Kohli', 'Raina', 'Dravid'])
print(df)

Output



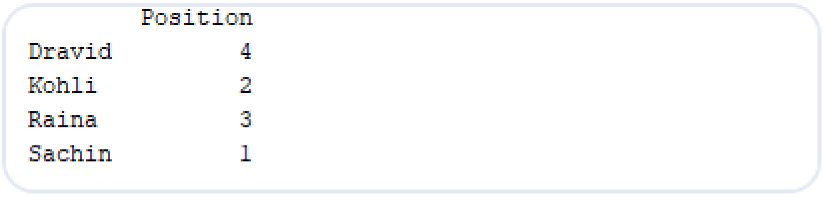
```
        Matches played  Position  Runrate
Dravid             NaN         4       50
Kohli            300.0         2       70
Raina            200.0         3       60
Sachin           400.0         1       80
```

Delete existing coulmns

We can delete columns using the **`delete` and `pop` functions**. For example to **delete the ‘Matches played’ column** in the above example, we can do it by either of the below 2 ways:

del df['Matches played']
or

df.pop('Matches played')



```
         Position
Dravid          4
Kohli           2
Raina           3
Sachin          1
```

Pandas Series
Pandas series is a one-dimensional data structure. It can hold data of many types including objects, floats, strings and integers. You can create a series by calling pandas.Series().
An list, numpy array, dict can be turned into a pandas series. You should use the simplest data structure that meets your needs. In this article we'll discuss the series data structure.
Create series

Introduction
Pandas comes with many data structures for processing data. One of them is a series.
The syntax for a series is:

```
import pandas as pd
s = pd.Series()
print(s)
```

This creates an empty series.
Create series from list
To turn a list into a series, all you have to do is:

```
>>> import pandas as pd
>>> items = [1,2,3,4]
>>> s = pd.Series(items)
```

The contents of s is:

```
0 1
1 2
2 3
3 4
dtype: int64
```

By default is assigns an index. First it shows the index, then the element value.
Create series from ndarray
You can create a series from a numpy ndarray.

```
import pandas as pd
import numpy as np
data = np.array(['x','y','z'])
s = pd.Series(data)
```

This ouputs the following:

```
>>> s
0   x
1   y
```

```
2    z
dtype: object
>>>
```

Create a series from a dict

If you have a dictionary, you can turn it into a series:

```
>>> import pandas as pd
>>> import numpy as np
>>> data = { 'uk':'united kingdom','fr':'france' }
>>> s = pd.Series(data)
```

The contents of the series is as follows:

```
>>> s
uk    united kingdom
fr      france
dtype: object
>>>
```

As index it used the dictionary keys.

Pandas series

Pandas series get index

You can access series data like you would with a list or ndarray.

```
>>> import pandas as pd
>>> import numpy as np
>>> data = np.array(['x','y','z'])
>>> s = pd.Series(data)
>>> s[0]
'x'
>>> s[1]
'y'
>>>
```

You slice a series, like you would with a list:

```
>>> data = np.array([1,2,3,4,5,6])
>>> s = pd.Series(data)
>>> s[:3]
0    1
1    2
2    3
dtype: int64
```

```
>>> s[3:5]
3    4
4    5
dtype: int64
>>>
```

Arithmetics with DataFrames and Series

Arithmetic operations are some of the most fundamental (and important) things you can do with **series and dataframes**.
We are interested in the following scenarios:

- Operations between series with the same index.
- Operations between dataframes with the same index.
- Operations between dataframe/series with the same index.
- Operations between series with different indexes.
- Operations between dataframes with different indexes.
- Operations between dataframe/series with different indexes.

1. Same index, obvious behavior

If two **(or more) series/dataframes share** the **same index** (both row and column index in the case of dataframes), operations follow the obvious **element-wise behavior** you would expect if you've used NumPy in the past:

```
import pandas as pd
ser_1 = pd.Series([1,2,3,4], index=['a', 'b', 'c', 'd'])
ser_2 = pd.Series([10,20,30,40], index=['a', 'b', 'c', 'd'])
print(ser_1)
print(ser_2)
```

output:

```
a   1
b   2
c   3
d   4
dtype: int64

a   10
b   20
c   30
d   40
dtype: int64
```

# Addition of two series with the same index

```
ser_1 + ser_2
a   11
b   22
c   33
d   44
dtype: int64
```

# Subtraction of two series with the same index

```
ser_2 - ser_1
a    9
b   18
c   27
d   36
dtype: int64
```

# Multiplication of two series with the same index

```
ser_1 * ser_2
a    10
b    40
c    90
d   160
dtype: int64
```

# Division of two series with the same index

```
ser_2 / ser_1
a   10.0
b   10.0
c   10.0
d   10.0
dtype: float64
```

The same behavior is shown when you apply operations on two dataframes that share both the row and column index:

```
import numpy as np
df_1 = pd.DataFrame(np.arange(1,17).reshape(4,4),
             index= ['Fi', 'Se', 'Th', 'Fo'],
             columns = ['a', 'b', 'c', 'd'])

df_2 = pd.DataFrame(np.arange(1,17).reshape(4,4) * 10,
             index= ['Fi', 'Se', 'Th', 'Fo'],
             columns = ['a', 'b', 'c', 'd'])
df_1
```

	a	**b**	**c**	**d**
Fi	1	2	3	4
Se	5	6	7	8
Th	9	10	11	12
Fo	13	14	15	16

df_2

	a	**B**	**c**	**d**
Fi	10	20	30	40
Se	50	60	70	80
Th	90	100	110	120
Fo	130	140	150	160

# Addition of two dataframes with the same index

df_1 + df_2

	a	**B**	**c**	**d**

	a	B	c	d
Fi	11	22	33	44
Se	55	66	77	88
Th	99	110	121	132
Fo	143	154	165	176

# Multiplication of two dataframes with the same index

df_1 * df_2

	a	B	c	d
Fi	10	40	90	160
Se	250	360	490	640
Th	810	1000	1210	1440
Fo	1690	1960	2250	2560

It's also possible to **perform operations between dataframes and series that share an index.** The default behavior is to align **the index of the series with the column index of the dataframe** and **perform the operations between each row and the series.**

# Sum a series and a dataframe

ser_1 + df_1

	a	b	c	d
Fi	2	4	6	8
Se	6	8	10	12
Th	10	12	14	16
Fo	14	16	18	20

Different index, outer joins

If you perform **operations between series/dataframes** with **different index**, the **result will be a new data structure whose index** is the **union of the original indexes**. If you have worked with databases before this is similar to an outer join using the indexes of the original series/dataframes. This is much easier to see with an example:

```
ser_1 = pd.Series([1,1,1,1,1], index=['a', 'b', 'c', 'd', 'e'])
ser_2 = pd.Series([5,5,5,5,5], index=['c', 'd', 'e', 'f', 'g'])

print(ser_1)
print(ser_2)
a   1
b   1
c   1
d   1
```

```
e    1
dtype: int64
c    5
d    5
e    5
f    5
g    5
dtype: int64
```

If the operation is performed on **series with different indexes**, the result will contain the **result of the operation on all entries whose index is contained in the union of the original indexes**. Elements **outside of the union will be filled with NaN.**
In this case, the union is ['c', 'd', 'e'].

```
ser_1 + ser_2
a    NaN
b    NaN
c    6.0
d    6.0
e    6.0
f    NaN
g    NaN
dtype: float64
ser_1 * ser_2
a    NaN
b    NaN
c    5.0
d    5.0
e    5.0
f    NaN
g    NaN
dtype: float64
```

Dataframes have the same behavior, but the unions are performed on both the row and column index.

```
import numpy as np
# In this case, the union are the elements [a,b,c] in the columns and [Fi,Fo,Th] in the rows
df_1 = pd.DataFrame(np.arange(1,17).reshape(4,4),
             index= ['Fi', 'Ma', 'Th', 'Fo'],
             columns = ['a', 'b', 'c', 'd'])

df_2 = pd.DataFrame(np.arange(1,17).reshape(4,4) * 10,
             index= ['Fi', 'Se', 'Th', 'Fo'],
             columns = ['a', 'b', 'c', 'e'])
df_1 + df_2
```

	a	**b**	**c**	**d**	**e**
Fi	11.0	22.0	33.0	NaN	NaN
Fo	143.0	154.0	165.0	NaN	NaN

	a	b	c	d	e
Ma	NaN	NaN	NaN	NaN	NaN
Se	NaN	NaN	NaN	NaN	NaN
Th	99.0	110.0	121.0	NaN	NaN

In the case of **operations between dataframes and series** with **different indexes, a union will be performed** between **the column index of the dataframe** and **the index of the series:**

df_1 + ser_2

	a	b	c	d	e	f	g
Fi	NaN	NaN	8.0	9.0	NaN	NaN	NaN
Ma	NaN	NaN	12.0	13.0	NaN	NaN	NaN
Th	NaN	NaN	16.0	17.0	NaN	NaN	NaN
Fo	NaN	NaN	20.0	21.0	NaN	NaN	NaN

Handling Missing Data in Pandas: NaN Values Explained

You have a couple of alternatives to work with missing data. You can:

- Drop the whole row
- Fill the row-column combination with some value

NaN means missing data

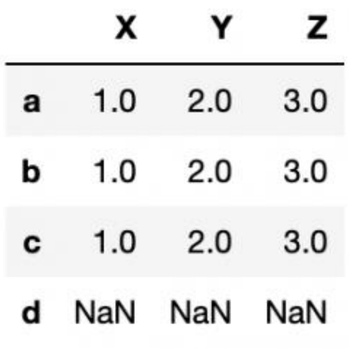
- Missing data is labelled **NaN.**

```
import pandas as pd
import numpy as np
df = pd.DataFrame([np.arange(1,4)],index=['a','b','c'],
columns=["X","Y","Z"])
```

	X	Y	Z
a	1	2	3
b	1	2	3
c	1	2	3

Now reindex this array adding an index **d**. Since d has no value it is filled with **NaN**.

```
df.reindex(index=['a','b','c','d'])
```

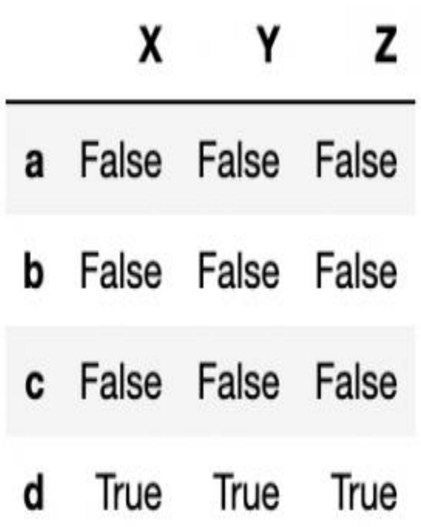


	X	Y	Z
a	1.0	2.0	3.0
b	1.0	2.0	3.0
c	1.0	2.0	3.0
d	NaN	NaN	NaN

isna

Now use **isna** to check for missing values.

```
pd.isna(df)
```

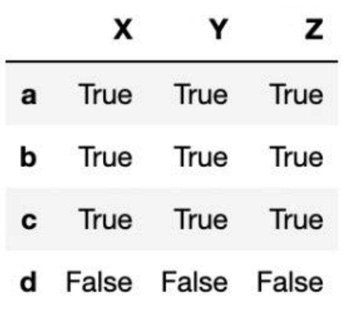


	X	Y	Z
a	False	False	False
b	False	False	False
c	False	False	False
d	True	True	True

notna

The opposite check—looking for actual values—is **notna()**.

```
pd.notna(df)
```

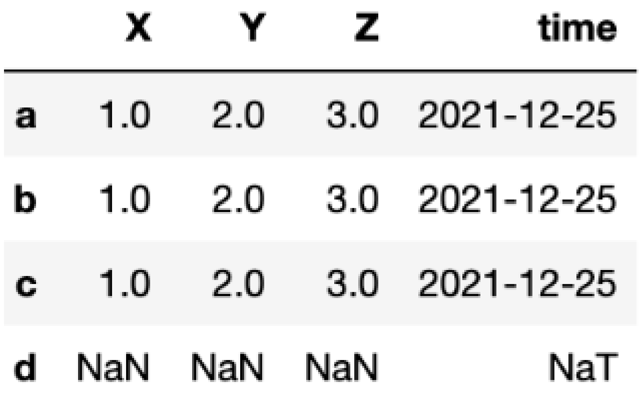


	X	Y	Z
a	True	True	True
b	True	True	True
c	True	True	True
d	False	False	False

nat

nat means a missing date.

```
df['time'] = pd.Timestamp('20211225')
df.loc['d'] = np.nan
```

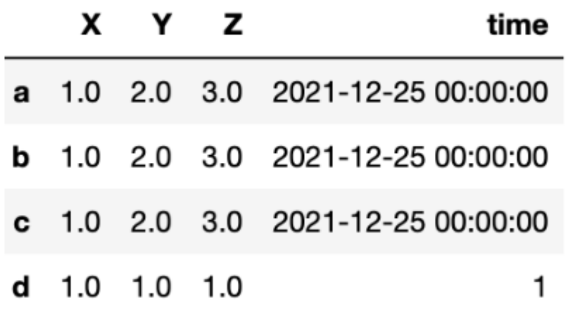


	X	Y	Z	time
a	1.0	2.0	3.0	2021-12-25
b	1.0	2.0	3.0	2021-12-25
c	1.0	2.0	3.0	2021-12-25
d	NaN	NaN	NaN	NaT

fillna

Here we can fill NaN values with the integer 1 using **fillna(1)**. The date column is not changed since the integer 1 is not a date.

```
df=df.fillna(1)
```



	X	Y	Z	time
a	1.0	2.0	3.0	2021-12-25 00:00:00
b	1.0	2.0	3.0	2021-12-25 00:00:00
c	1.0	2.0	3.0	2021-12-25 00:00:00
d	1.0	1.0	1.0	1

To fix that, fill empty time values with:

```
df['time'].fillna(pd.Timestamp('20221225'))
```

Filling out Missing Values

- Fill NA with Mean, Median or Mode of the data
- Fill NA with a constant value
- Forward Fill or Backward Fill NA
- Interpolate Data and Fill NA

Fill Missing DataFrame Values with Column Mean, Median and Mode

`fillna()` method fills the NA-marked values with values you supply the method with.

For example, you can use the `.median()`, `.mode()` and `.mean()` functions on a column, and supply those as the fill value:

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

# Using median
df['Salary'].fillna(df['Salary'].median(), inplace=True)

# Using mean
df['Salary'].fillna(int(df['Salary'].mean()), inplace=True)

# Using mode
df['Salary'].fillna(int(df['Salary'].mode()), inplace=True)
```

Fill Missing DataFrame Values with a Constant

You could also decide to fill the **NA-marked values** with a **constant value**. For example, you can put in a special string or numerical value:

```
df['Salary'].fillna(0, inplace=True)
```

Forward Fill Missing DataFrame Values

This method would fill the missing values with first non-missing value that occurs before it:

```
df['Salary'].fillna(method='ffill', inplace=True)
```

Backward Fill Missing DataFrame Values

This method would fill the missing values with first ***non-missing value that occurs after it***:

```
df['Salary'].fillna(method='bfill', inplace=True)
```

Fill Missing DataFrame Values with Interpolation

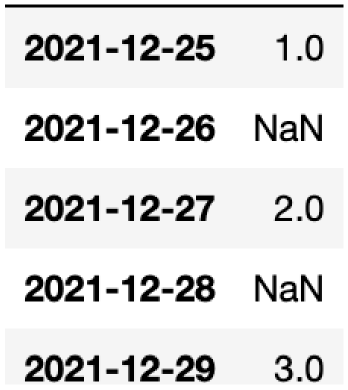
Finally, this method uses mathematical interpolation to determine what value would have been in the place of a missing value:

```
df['Salary'].interpolate(method='polynomial')
```

interpolate

Another feature of Pandas is that **it will fill in missing values using what is logical.**
Consider a time series—let's say you're **monitoring some machine and on certain days it fails to report. Below it reports o**n **Christmas and every other day that week.** Then we reindex the Pandas Series, creating gaps in our timeline.

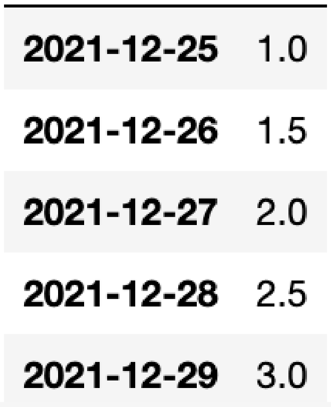
```
import pandas as pd
import numpy as np
arr=np.array([1,2,3])
idx=np.array([pd.Timestamp('20211225'),
pd.Timestamp('20211227'),
pd.Timestamp('20211229')])
df = pd.DataFrame(arr,index=idx)
idx=[pd.Timestamp('20211225'),
pd.Timestamp('20211226'),
pd.Timestamp('20211227'),
pd.Timestamp('20211228'),
pd.Timestamp('20211229')]
df=df.reindex(index=idx)
```



```
2021-12-25   1.0
2021-12-26   NaN
2021-12-27   2.0
2021-12-28   NaN
2021-12-29   3.0
```

We use the **interpolate()** function. Pandas fill them in nicely using **the midpoints between the points.** Of course, if this was curvilinear it would fit a function to that and find the average another way.

```
df=df.interpolate()
```



2021-12-25	1.0
2021-12-26	1.5
2021-12-27	2.0
2021-12-28	2.5
2021-12-29	3.0

dropna()

You can control whether you want to remove the rows containing at least 1 `NaN` or all `NaN` values by setting the `how` parameter in the `dropna` method.

how : {'any', 'all'}

- any: if any NA values are present, drop that label
- all: if all values are NA, drop that lab**el**

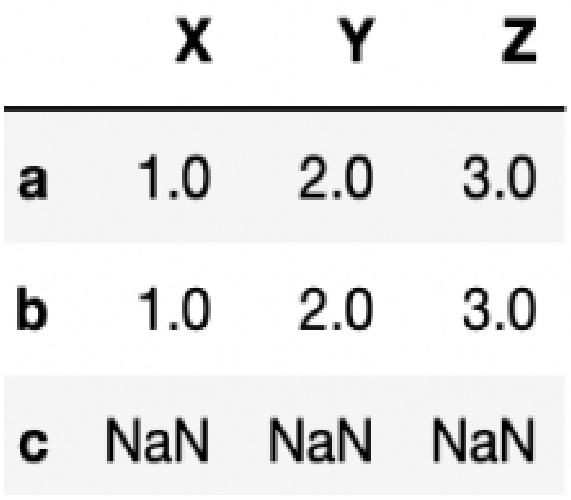
df.dropna(axis=0,inplace=True, how='all')

This would only remove the last row from the dataset since how=all would only drop a **row if all of the values are missing from the row.**

Similarly, to **drop columns containing missing values**, just set axis=1 in the dropna method.

dropna() means to drop rows or columns whose value is empty. Another way to say that is to show only rows or columns that are not empty.
Here we fill row c with **NaN**:

```
df = pd.DataFrame([np.arange(1,4)],index=['a','b','c'],
columns=["X","Y","Z"])
df.loc['c']=np.NaN
```



	X	Y	Z
a	1.0	2.0	3.0
b	1.0	2.0	3.0
c	NaN	NaN	NaN

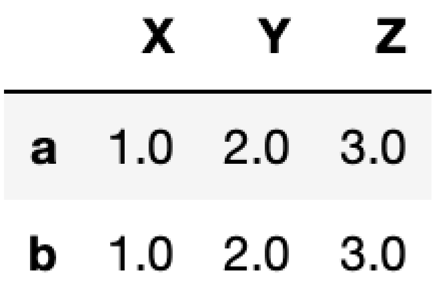
Then run **dropna** over the row (axis=0) axis.

```
df.dropna()
```

You could also write:

```
df.dropna(axis=0)
```

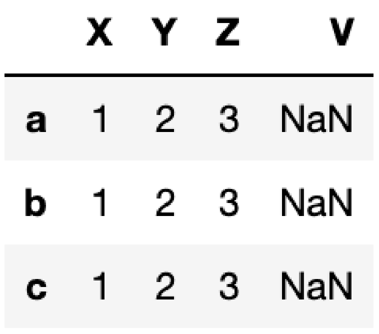
All rows except c were dropped:



	X	Y	Z
a	1.0	2.0	3.0
b	1.0	2.0	3.0

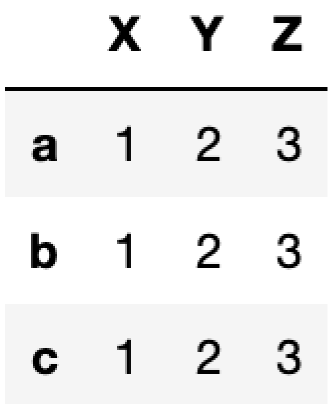
To drop the column:

```
df = pd.DataFrame([np.arange(1,4)],index=['a','b','c'],
columns=["X","Y","Z"])
df['V']=np.NaN
```



	X	Y	Z	V
a	1	2	3	NaN
b	1	2	3	NaN
c	1	2	3	NaN

```
df.dropna(axis=1)
```



	X	Y	Z
a	1	2	3
b	1	2	3
c	1	2	3