What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, 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 analyze 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.

**Data Science:** is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

* Is there a correlation between two or more columns?
* What is average value?
* Max value?
* Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.

## Where is the Pandas Codebase?

The source code for Pandas is located at this github repository <https://github.com/pandas-dev/pandas>

# **Pandas Getting Started**

## Installation of Pandas

If you have [Python](https://www.w3schools.com/python/default.asp) and [PIP](https://www.w3schools.com/python/python_pip.asp) already installed on a system, then installation of Pandas is very easy.

Install it using this command:

C:\Users\Your Name> pip install pandas

If this command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.

Import Pandas

Once Pandas is installed, import it in your applications by adding the import keyword:

import pandas

Now Pandas is imported and ready to use.

-------------------------------------------------------------------

import pandas  
mydataset = {  
  'cars': ["BMW", "Volvo", "Ford"],  
  'passings': [3, 7, 2]  
}

myvar = pandas.DataFrame(mydataset)  
print(myvar)

Benefits of Pandas

The benefits of pandas over using other language are as follows:

* **Data Representation:** It represents the data in a form that is suited for data analysis through its DataFrame and Series.
* **Clear code:** The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

## What is a Series?

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

### **Example**

Create a simple Pandas Series from a list:

import pandas as pd  
  
a = [1, 7, 2]  
  
myvar = pd.Series(a)  
  
print(myvar)

## Labels:

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

## Create Labels

With the index argument, you can name your own labels.

### **Example**

Create your own labels:

import pandas as pd  
  
a = [1, 7, 2]  
  
myvar = pd.Series(a, index = ["x", "y", "z"])  
  
print(myvar)

## Python Pandas Data Structure

The Pandas provides two data structures for processing the data, i.e., **Series** and **DataFrame**, which are discussed below:

### **1) Series**

It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the **index**. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:

**Data:** It can be any list, dictionary, or scalar value.

**Creating Series from Array:**

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

**import** pandas as pd

**import** numpy as np

info = np.array(['P','a','n','d','a','s'])

a = pd.Series(info)

**print**(a)

## Creating a Series:

We can create a Series in two ways:

1. Create an empty Series
2. Create a Series using inputs.

### **Create an Empty Series:**

We can easily create an empty series in Pandas which means it will not have any value.

The syntax that is used for creating an Empty Series:

1. series object> = pandas.Series()

The below example creates an Empty Series type object that has no values and having default datatype, i.e., **float64**.

**Example**

**import** pandas as pd

x = pd.Series()

print (x)

### **Creating a Series using inputs:**

We can create Series by using various inputs:

* Array
* Dict
* Scalar value

**Creating Series from Array:**

Before creating a Series, firstly, we have to import the **numpy** module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default index of **range(n)** is being passed where n defines the length of an array, i.e., [0,1,2,....**range(len(array))-1**].

**Example**

**import** pandas as pd

**import** numpy as np

info = np.array(['P','a','n','d','a','s'])

a = pd.Series(info)

print(a)

 if index is passed, then values correspond to a particular label in the index will be extracted from the **dictionary**.

#**import** the pandas library

**import** pandas as pd

**import** numpy as np

info = {'x' : 0., 'y' : 1., 'z' : 2.}

a = pd.Series(info)

print (a)

**Create a Series using Scalar:**

If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

#**import** pandas library

**import** pandas as pd

**import** numpy as np

x = pd.Series(4, index=[0, 1, 2, 3])

print (x)

### **Example:**

Create a Series using only data from "day1" and "day2":

import pandas as pd  
  
calories = {"day1": 420, "day2": 380, "day3": 390}  
  
myvar = pd.Series(calories, index = ["day1", "day2"])  
  
print(myvar)

Accessing data from series with Position:

Once you create the Series type object, you can access its indexes, data, and even individual elements.

The data in the Series can be accessed similar to that in the ndarray.

**import** pandas as pd

x = pd.Series([1,2,3],index = ['a','b','c'])

#retrieve the first element

print (x[0])

### **Series object attributes**

The Series attribute is defined as any information related to the Series object such as size, datatype. etc. Below are some of the attributes that you can use to get the information about the Series object:

| **Attributes** | **Description** |
| --- | --- |
| **Series.index** | Defines the index of the Series. |
| **Series.shape** | It returns a tuple of shape of the data. |
| **Series.dtype** | It returns the data type of the data. |
| **Series.size** | It returns the size of the data. |
| **Series.empty** | It returns True if Series object is empty, otherwise returns false. |
| **Series.hasnans** | It returns True if there are any NaN values, otherwise returns false. |
| **Series.nbytes** | It returns the number of bytes in the data. |
| **Series.ndim** | It returns the number of dimensions in the data. |
| **Series.itemsize** | It returns the size of the datatype of item. |

### **Retrieving Index array and data array of a series object**

We can retrieve the index array and data array of an existing Series object by using the attributes index and values.

**import** numpy as np

**import** pandas as pd

x=pd.Series(data=[2,4,6,8])

y=pd.Series(data=[11.2,18.6,22.5], index=['a','b','c'])

print(x.index)

print(x.values)

print(y.index)

print(y.values)

### **Retrieving Types (dtype) and Size of Type (itemsize)**

You can use attribute dtype with Series object as <objectname> dtype for retrieving the data type of an individual element of a series object, you can use the **itemsize** attribute to show the number of bytes allocated to each data item.

**import** numpy as np

**import** pandas as pd

a=pd.Series(data=[1,2,3,4])

b=pd.Series(data=[4.9,8.2,5.6],

index=['x','y','z'])

print(a.dtype)

print(a.itemsize)

print(b.dtype)

print(b.itemsize)

## DataFrames

Data sets in Pandas are usually multi-dimensional tables, called DataFrames.

Series is like a column, a DataFrame is the whole table.

DataFrame is defined as a standard way to store data that has two different indexes, i.e., **row index** and **column index**. It consists of the following properties:

* The columns can be heterogeneous types like int, bool, and so on.
* It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

## Parameter & Description:

**data:** It consists of different forms like ndarray, series, map, constants, lists, array.

**index:** The Default np.arrange(n) index is used for the row labels if no index is passed.

**columns:** The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.

**dtype:** It refers to the data type of each column.

**copy():** It is used for copying the data.

Create a DataFrame

We can create a DataFrame using following ways:

* **dict**
* **Lists**
* **Numpy ndarrrays**
* **Series**

**Create an empty DataFrame**

The below code shows how to create an empty DataFrame in Pandas:

# importing the pandas library

**import** pandas as pd

df = pd.DataFrame()

**print** (df)

**Explanation:** In the above code, first of all, we have imported the pandas library with the alias **pd** and then defined a variable named as **df** that consists an empty DataFrame. Finally, we have printed it by passing the **df** into the **print**.

### **Create a DataFrame using List:**

We can easily create a DataFrame in Pandas using list.

# importing the pandas library

**import** pandas as pd

# a list of strings

x = ['Python', 'Pandas']

# Calling DataFrame constructor on list

df = pd.DataFrame(x)

**print**(df)

### **Create a DataFrame from Dict of ndarrays/ Lists**

# importing the pandas library

**import** pandas as pd

info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}

df = pd.DataFrame(info)

**print** (df)

### **Create a DataFrame from Dict of Series:**

# importing the pandas library

**import** pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),

   'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)

**print** (d1)

Column Selection

We can select any column from the DataFrame. Here is the code that demonstrates how to select a column from the DataFrame.

# importing the pandas library

**import** pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),

   'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)

**print** (d1 ['one'])

-------------------------------------------------------------------------------------

Custom indexing Rearranging the column accessing values loc(),iloc().

The Pandas offers .loc[] and .iloc[] methods for **data slicing**. Data Slicing generally refers to inspect your data sets. These two methods belong to the index selection method that is used to set an identifier for each row of the data set. The indexing can take specific labels, and these labels can either be an integer or any other value specified by the user.

The .**loc[]** method is used to retrieve the group of rows and columns by labels or a boolean array present in the DataFrame. It takes only index labels, and if it exists in the caller DataFrame, it returns the rows, columns, or DataFrame. It is a label-based method but may be used with the boolean array.

Whereas, the **.iloc[]** method is used when the index label of the DataFrame is other than numeric series of 0,1,2,....,n, or in the case when the user does not know the index label.

# importing the module

**import** pandas as pd

# creating a sample dataframe

data **=** pd.DataFrame({'Brand': ['Maruti', 'Hyundai', 'Tata',

                               'Mahindra', 'Maruti', 'Hyundai',

                               'Renault', 'Tata', 'Maruti'],

                     'Year': [2012, 2014, 2011, 2015, 2012,

                              2016, 2014, 2018, 2019],

                     'Kms Driven': [50000, 30000, 60000,

                                    25000, 10000, 46000,

                                    31000, 15000, 12000],

                     'City': ['Gurgaon', 'Delhi', 'Mumbai',

                              'Delhi', 'Mumbai', 'Delhi',

                              'Mumbai', 'Chennai',  'Ghaziabad'],

                     'Mileage':  [28, 27, 25, 26, 28,

                                  29, 24, 21, 24]})

# displaying the DataFrame

display(data)

===========================================

## Python loc() function

The[loc() function](https://www.geeksforgeeks.org/python-pandas-dataframe-loc/) is label based data selecting method which means that we have to pass the name of the row or column which we want to select. This method includes the last element of the range passed in it, unlike iloc(). loc() can accept the boolean data unlike iloc(). Many operations can be performed using the loc() method like

### Example 1:

Selecting data according to some conditions

| # selecting cars with brand 'Maruti' and Mileage > 25.  display(data.loc[(data.Brand **==** 'Maruti') & (data.Mileage >25)]) |
| --- |

### Example 2:

Selecting a range of rows from the DataFrame

# selecting range of rows from 2 to 5

display(data.loc[2: 5])

### Example 3:

Updating the value of any column

# updating values of Mileage if Year < 2015

data.loc[(data.Year < 2015), ['Mileage']] **=** 22

display(data)

## Python iloc() function

The[iloc() function](https://www.geeksforgeeks.org/python-extracting-rows-using-pandas-iloc/) is an indexed-based selecting method which means that we have to pass an integer index in the method to select a specific row/column. This method does not include the last element of the range passed in it unlike loc(). iloc() does not accept the boolean data unlike loc(). Operations performed using iloc() are:

### Example 1:

Selecting rows using integer indices

| # selecting 0th, 2nd, 4th, and 7th index rows  display(data.iloc[[0, 2, 4, 7]]) |
| --- |

### Example 2:

Selecting a range of columns and rows simultaneously

| # selecting rows from 1 to 4 and columns from 2 to 4  display(data.iloc[1: 5, 2: 5]) |
| --- |

# Adding new column to existing DataFrame in Pandas

# Import pandas package

**import** pandas as pd

# Define a dictionary containing Students data

data **=** {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],

        'Height': [5.1, 6.2, 5.1, 5.2],

        'Qualification': ['Msc', 'MA', 'Msc', 'Msc']}

# Convert the dictionary into DataFrame

df **=** pd.DataFrame(data)

# Declare a list that is to be converted into a column

address **=** ['Delhi', 'Bangalore', 'Chennai', 'Patna']

# Using 'Address' as the column name

# and equating it to the list

df['Address'] **=** address

# Observe the result

print(df)

**Method #2:** By using dataframe.insert().

It gives the freedom to add a column at any position we like and not just at the end. It also provides different options for inserting the column values.

# Import pandas package

**import** pandas as pd

# Define a dictionary containing Students data

data **=** {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],

        'Height': [5.1, 6.2, 5.1, 5.2],

        'Qualification': ['Msc', 'MA', 'Msc', 'Msc']}

# Convert the dictionary into DataFrame

df **=** pd.DataFrame(data)

# Using DataFrame.insert() to add a column

df.insert(2, "Age", [21, 23, 24, 21], True)

# Observe the result

print(df)

-------------------------------------------------------------------

To **Delete a column from a Pandas DataFrame** or **Drop one or more than one column from a DataFrame** can be achieved in multiple ways.

Create a simple Dataframe with dictionary of lists, say column names are **A, B, C, D, E**. In this article, we will cover 6 different methods to delete some columns from Pandas DataFrame.

# Import pandas package

**import** pandas as pd

# create a dictionary with five fields each

data **=** {

    'A': ['A1', 'A2', 'A3', 'A4', 'A5'],

    'B': ['B1', 'B2', 'B3', 'B4', 'B5'],

    'C': ['C1', 'C2', 'C3', 'C4', 'C5'],

    'D': ['D1', 'D2', 'D3', 'D4', 'D5'],

    'E': ['E1', 'E2', 'E3', 'E4', 'E5']}

# Convert the dictionary into DataFrame

df **=** pd.DataFrame(data)

print(df)

### Method 1: Drop Columns from a Dataframe using drop method.

**Example 1:** **Remove specific single columns.**

# Import pandas package

**import** pandas as pd

# create a dictionary with five fields each

data **=** {

    'A': ['A1', 'A2', 'A3', 'A4', 'A5'],

    'B': ['B1', 'B2', 'B3', 'B4', 'B5'],

    'C': ['C1', 'C2', 'C3', 'C4', 'C5'],

    'D': ['D1', 'D2', 'D3', 'D4', 'D5'],

    'E': ['E1', 'E2', 'E3', 'E4', 'E5']}

# Convert the dictionary into DataFrame

df **=** pd.DataFrame(data)

# Remove column name 'A'

Print(df.drop(['A'], axis**=**1))

**Example 2: Remove specific multiple columns.**

| # Import pandas package  **import** pandas as pd    # create a dictionary with five fields each  data **=** {      'A': ['A1', 'A2', 'A3', 'A4', 'A5'],      'B': ['B1', 'B2', 'B3', 'B4', 'B5'],      'C': ['C1', 'C2', 'C3', 'C4', 'C5'],      'D': ['D1', 'D2', 'D3', 'D4', 'D5'],      'E': ['E1', 'E2', 'E3', 'E4', 'E5']}    # Convert the dictionary into DataFrame  df **=** pd.DataFrame(data)    # Remove two columns name is 'C' and 'D'  df.drop(['C', 'D'], axis**=**1)    # df.drop(columns =['C', 'D']) |
| --- |

**Example 3: Remove columns based on column index.**

| # Import pandas package  **import** pandas as pd    # create a dictionary with five fields each  data **=** {      ‘A’: [‘A1’, ‘A2’, ‘A3’, ‘A4’, ‘A5’],      ‘B’: [‘B1’, ‘B2’, ‘B3’, ‘B4’, ‘B5’],      ‘C’: [‘C1’, ‘C2’, ‘C3’, ‘C4’, ‘C5’],      ‘D’: [‘D1’, ‘D2’, ‘D3’, ‘D4’, ‘D5’],      ‘E’: [‘E1’, ‘E2’, ‘E3’, ‘E4’, ‘E5’]}    # Convert the dictionary into DataFrame  df **=** pd.DataFrame(data)    # Remove three columns as index base  df.drop(df.columns[[0, 4, 2]], axis**=**1, inplace**=**True)    print(df) |
| --- |

In boolean indexing, we will select subsets of data based on the actual values of the data in the DataFrame and not on their row/column labels or integer locations. In boolean indexing, we use a boolean vector to filter the data.

Boolean indexing is a type of indexing that uses actual values of the data in the DataFrame. In boolean indexing, we can filter a data in four ways:

* Accessing a DataFrame with a boolean index
* Applying a boolean mask to a dataframe
* Masking data based on column value
* Masking data based on an index value

### Accessing a DataFrame with a boolean index:

In order to access a dataframe with a boolean index, we have to create a dataframe in which the index of dataframe contains a boolean value that is “True” or “False”.

**Example** :

| # importing pandas as pd  **import** pandas as pd    # dictionary of lists  dict **=** {'name':["aparna", "pankaj", "sudhir", "Geeku"],          'degree': ["MBA", "BCA", "M.Tech", "MBA"],          'score':[90, 40, 80, 98]}    df **=** pd.DataFrame(dict, index **=** [True, False, True, False])    print(df) |
| --- |

If the data is stored in the form of rows and columns or two-dimensional data is generally called dataframes in pandas.

If we have two dataframes, then with the help of pandas, we can combine them or merge them into a single dataframe. Pandas provide the set logic to combine the data of two different dataframes as well as logic to compare them.

### **1. Using concat() function**

In python, we can concatenate the two dataframes with the help of the concat() function of Pandas. We can concatenate the data either row-wise or column-wise. This function merges the data on one axis (row or column) and performs the set logic on another axis (another index).

**Example:**

**import** pandas as pd

**from** IPython.display **import** display

# First DataFrame

dataFrame1 = pd.DataFrame({'id': ['A1', 'A2', 'A3', 'A4'],

                    'Name': ['ABC', 'PQR', 'DEF', 'GHI'],

                    'Marks':[65,69,96,89]})

# Second DataFrame

dataFrame2 = pd.DataFrame({'id': ['B1', 'B2', 'B3', 'B4'],

                    'Name': ['XYZ', 'TUV', 'MNO', 'JKL'],

                    'Marks':[56,96,69,98]})

frames = [dataFrame1, dataFrame2]

result = pd.concat(frames)

print(result)

### **2. Using joins in pandas**

We have understood the concept of joins in the database where we join the two tables based on some common attribute. The same method is applicable in the concatenation of dataframes. In the simple concat() method, we merged all the rows on one another and created the new dataframe. In the join, we define which type of join we want to perform on the table, whether it is an inner join or an outer join. Whatever type of join either inner join (intersection) or outer join (union), will be defined in the join attribute.

**Example:**

**import** pandas as pd

**from** IPython.display **import** display

dataFrame1 = pd.DataFrame({'id': ['A1', 'A2', 'A3', 'A4'],

                    'Name': ['ABC', 'PQR', 'TUV', 'JKL']})

dataFrame2 = pd.DataFrame({'City': ['NOIDA', 'JAIPUR', 'MANALI', 'DELHI'],

                    'Age': ['11', '10', '12', '17']})

# the default behaviour is join='outer'

# inner join

result = pd.concat([dataFrame1, dataFrame2], axis=1, join='inner')

print(result)

**Example:**

**import** pandas as pd

**from** IPython.display **import** display

dataFrame1 = pd.DataFrame({'id': ['A1', 'A2', 'A3', 'A4'],

                    'Name': ['ABC', 'PQR', 'TUV', 'JKL']})

dataFrame2 = pd.DataFrame({'id': ['B1', 'B2', 'B3', 'B4'],

                           'City': ['NOIDA', 'JAIPUR', 'MANALI', 'DELHI'],

                           'Age': ['11', '10', '12', '17']})

# the default behaviour is join='outer'

# inner join

result = pd.concat([dataFrame1, dataFrame2], axis=0, join='inner')

display(result)

==============================================

# Grouping and Aggregating with Pandas:

Grouping and aggregating will help to achieve data analysis easily using various functions. These methods will help us to the group and summarize our data and make complex analysis comparatively easy.

**Creating a sample dataset of marks of various subjects.**

| # import module  **import** pandas as pd    # Creating our dataset  df **=** pd.DataFrame([[9, 4, 8, 9],                     [8, 10, 7, 6],                     [7, 6, 8, 5]],                    columns**=**['Maths',  'English',                             'Science', 'History'])    # display dataset  print(df) |
| --- |

**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. The function used for aggregation is agg(), the parameter is the function we want to perform.

Some functions used in the aggregation are:

***Function Description:***

* *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*

**Examples:**

* The sum() function is used to calculate the sum of every value.

**import** pandas as pd

# Creating our dataset

df **=** pd.DataFrame([[9, 4, 8, 9],

                   [8, 10, 7, 6],

                   [7, 6, 8, 5]],

                  columns**=**['Maths',  'English',

                           'Science', 'History'])

# display dataset

print(df)

df.sum()

The describe() function is used to get a summary of our dataset.

| df.describe() |
| --- |

We used agg() function to calculate the sum, min, and max of each column in our dataset.

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

**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()

------------------------------------------------------------------------------------------------

# Working with Missing Data in Pandas

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 a real-life scenarios. Missing Data can also refer to as NA(Not Available) values in pandas. In DataFrame sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. 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.

In Pandas missing data is represented by two value:

* None: None is a Python singleton object that is often used for missing data in Python code.
* NaN : NaN (an acronym for Not a Number), is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

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()

#### Checking for missing values using isnull()

In order to check null values in Pandas DataFrame, we use isnull() function this function return dataframe of Boolean values which are True for NaN values.

# 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, 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

print(df.isnull())

# importing pandas as pd

# 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, 45, 56, np.nan],

'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe using dictionary

df = pd.DataFrame(dict)

print(df)

# using notnull() function

print(df.notnull())

------------------------------------------------------------------------------------------------

### 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.

# 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, 45, 56, np.nan],

'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from dictionary

df = pd.DataFrame(dict)

print(df)

# filling missing value using fillna()

print(df.fillna(0))

# 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, 45, 56, np.nan],

'Third Score':[np.nan, 40, 80, 98]}

# creating a dataframe from dictionary

df = pd.DataFrame(dict)

print(df)

# filling a missing value with

# previous ones

print(df.fillna(method ='pad'))

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Working with missing data reading data from csv,excel,json.

## 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'.

import pandas as pd

df = pd.read\_csv('C:\\Users\\eluri\\OneDrive\\Desktop\\jos\\data.csv')

print(df)

# How to Read JSON Files with Pandas?

What is json?

Stands for javascript object notation

Lightweight format for storing and transporting data

Often used when data is sent from a server to a web page

Consists of attribute-value pairs and array datatypes

Syntax Of Json:--

{

“Name”:{“0”:”Angel”,”1”:”john”},

“Age”:{“0”:”30”,”1”:”14”},

“Score”:{“0”:”4”,”1”:”9”}

}

import pandas as pd

df = pd.read\_json("FILE\_JSON.json")

df.head()

===========================================================================

## Reading JSON Files using Pandas

To read the files, we use**read\_json()**function and through it, we pass the path to the JSON file we want to read. Once we do that, it returns a “DataFrame”( A table of rows and columns) that stores data. If we want to read a file that is located on remote servers then we pass the link to its location instead of a local path.

==========================================================================

import pandas as pd

data = {

"One": {

"0": 60,

"1": 60,

"2": 60,

"3": 45,

"4": 45,

"5": 60

},

"Two": {

"0": 110,

"1": 117,

"2": 103,

"3": 109,

"4": 117,

"5": 102

}

}

df = pd.DataFrame(data)

print(df)

Writing data to csv,excel,json,html,reading data from database and storing in data frame.

# importing packages

import pandas as pd

# dictionary of data

dct = {'ID': {0: 23, 1: 43, 2: 12,

3: 13, 4: 67, 5: 89,

6: 90, 7: 56, 8: 34},

'Name': {0: 'Ram', 1: 'Deep',

2: 'Yash', 3: 'Aman',

4: 'Arjun', 5: 'Aditya',

6: 'Divya', 7: 'Chalsea',

8: 'Akash' },

'Marks': {0: 89, 1: 97, 2: 45, 3: 78,

4: 56, 5: 76, 6: 100, 7: 87,

8: 81},

'Grade': {0: 'B', 1: 'A', 2: 'F', 3: 'C',

4: 'E', 5: 'C', 6: 'A', 7: 'B',

8: 'B'}

}

# forming dataframe

data = pd.DataFrame(dct)

# storing into the excel file

data.to\_excel("C:\\Users\\eluri\\OneDrive\\Desktop\\jos\\output.xlsx")

Writing data frame to database handling pdf files.

Matplotlib:

Basic plotting

i.colors

ii.styles

iii.seaborn themes

c. labels

d.title

e.legend

f.axis

g.bar chart

h.histogram

scatter plot

box plot

pie chart