

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.

Key Features of Pandas

- Fast and efficient DataFrame 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 subsetting 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.

andas 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.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Building and handling two or more dimensional arrays is a tedious task, burden is placed on the user to consider the orientation of the data set when writing functions. But using Pandas data structures, the mental effort of the user is reduced.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1.

Mutability

All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.

Note – DataFrame is widely used and one of the most important data structures. Panel is used much less.

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,

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

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.

Data Type of Columns

The data types of the four columns are as follows –

Column	Type
Name	String
Age	Integer
Gender	String
Rating	Float

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

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

pandas.Series

A pandas Series can be created using the following constructor –

```
pandas.Series( data, index, dtype, copy)
```

The parameters of the constructor are as follows –

Sr.No	Parameter & Description
1	data data takes various forms like ndarray, list, constants
2	index Index values must be unique and hashable, same length as data. Default np.arange(n) if no index is passed.
3	dtype dtype is for data type. If None, data type will be inferred
4	copy Copy data. Default False

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

Create an Empty Series

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

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
```

```
print s
```

Its **output** is as follows –

```
Series([], dtype: float64)
```

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**].

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
0  a
1  b
2  c
3  d
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

Example 2

```
#import the pandas library and aliasing as pd
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
```

Its **output** is as follows –

```
100 a
101 b
102 c
103 d
dtype: object
```

We passed the index values here. Now we can see the customized indexed values in the output.

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.

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
a 0.0
b 1.0
c 2.0
dtype: float64
```

Observe – Dictionary keys are used to construct index.

Example 2

```
#import the pandas library and aliasing as pd
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
```

Its **output** is as follows –

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe – Index order is persisted and the missing element is filled with NaN (Not a Number).

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 the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print s
```

Its **output** is as follows –

```
0 5
1 5
2 5
3 5
dtype: int64
```

Accessing Data from Series with Position

Data in the series can be accessed similar to that in an **ndarray**.

Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

```
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]
```

Its **output** is as follows –

1

Example 2

Retrieve the first three elements in the Series. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index)

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print s[:3]
```

Its **output** is as follows –

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

Example 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:]
```

Its **output** is as follows –

```
c 3  
d 4  
e 5  
dtype: int64
```

Retrieve Data Using Label (Index)

A Series is like a fixed-size **dict** in that you can get and set values by index label.

Example 1

Retrieve a single element using index label value.

```
import pandas as pd  
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
```

```
#retrieve a single element  
print s['a']
```

Its **output** is as follows –

```
1
```

Example 2

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']]
```

Its **output** is as follows –

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

Example 3

If a label is not contained, an exception is raised.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
```

```
#retrieve multiple elements
print s['f']
```

Its **output** is as follows –

```
...
KeyError: 'f'
```

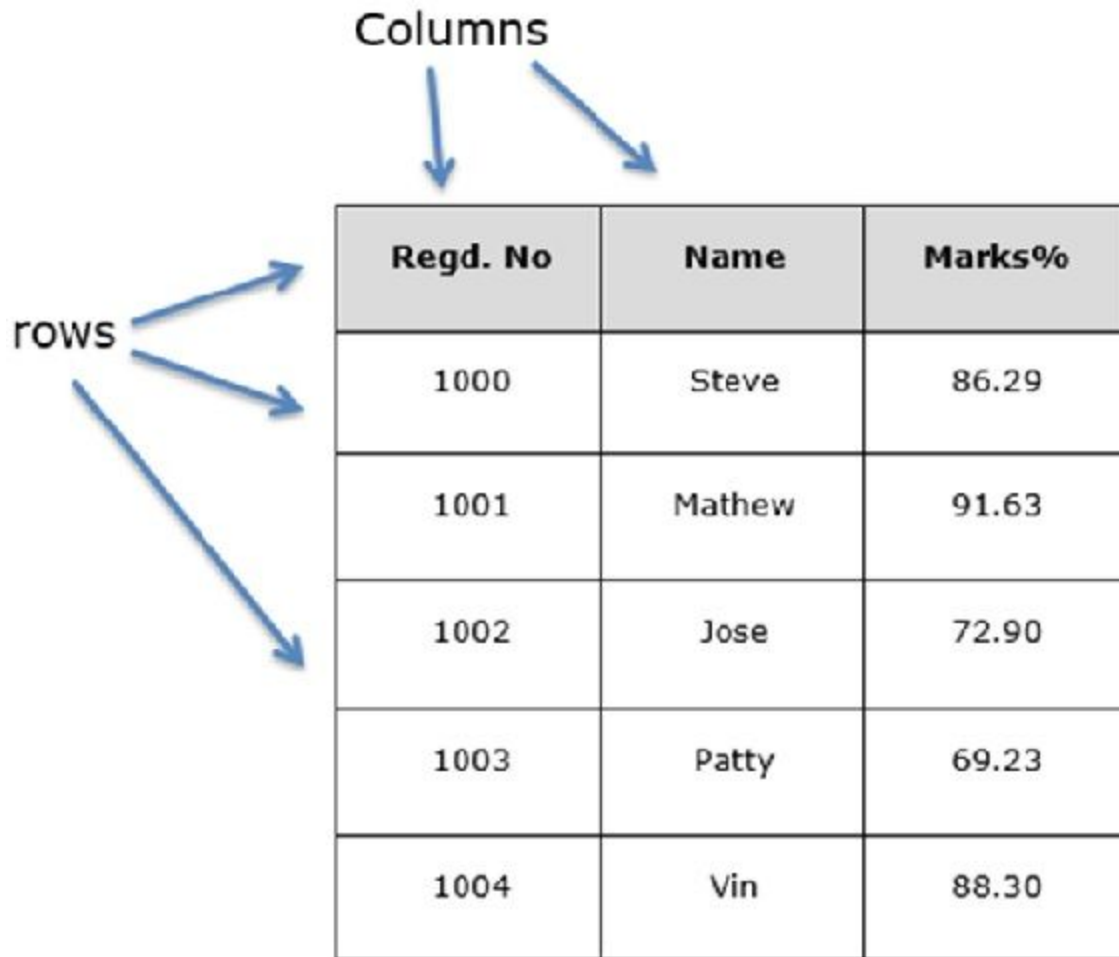
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

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.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor –

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

The parameters of the constructor are as follows –

Sr.No	Parameter & Description
-------	-------------------------

1

data

data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.

2

index

For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.

3

columns

For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.

4

dtype

Data type of each column.

5

copy

This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like –

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print df
```

Its **output** is as follows –

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print df
```

Its **output** is as follows –

```
0
0  1
1  2
2  3
3  4
4  5
```

Example 2

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print df
```

Its **output** is as follows –

```
   Name  Age
```

```
0  Alex    10
1  Bob     12
2  Clarke  13
```

Example 3

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print df
```

Its **output** is as follows –

```
   Name  Age
0  Alex  10.0
1   Bob  12.0
2 Clarke  13.0
```

Note – Observe, the **dtype** parameter changes the type of Age column to floating point.

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.

Example 1

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print df
```

Its **output** is as follows –

```
   Age  Name
0   28   Tom
1   34  Jack
2   29 Steve
3   42  Ricky
```

Note – Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example 2

Let us now 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
```

Its **output** is as follows –

	Age	Name
rank1	28	Tom
rank2	34	Jack
rank3	29	Steve
rank4	42	Ricky

Note – Observe, the **index** parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print df
```

Its **output** is as follows –

a	b	c
---	---	---

```
0  1  2   NaN
1  5 10 20.0
```

Note – Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print df
```

Its **output** is as follows –

```
   a  b   c
first 1  2  NaN
second 5 10 20.0
```

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
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=['a', 'b1'])
print df1
print df2
```

Its **output** is as follows –

```
#df1 output
```



```

      a b
first  1 2
second 5 10

#df2 output
      a b1
first  1 NaN
second 5 NaN

```

Note – Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```

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

```

Its **output** is as follows –

```

      one  two
a    1.0    1
b    2.0    2
c    3.0    3
d    NaN    4

```

Note – Observe, for the series one, there is no label 'd' passed, but in the result, for the d label, NaN is appended with NaN.

Let us now understand **column selection**, **addition**, and **deletion** through examples.

Column Selection

We will understand this by selecting a column from the DataFrame.

Example

```
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']
```

Its **output** is as follows –

```
a    1.0
b    2.0
c    3.0
d     NaN
Name: one, dtype: float64
```

Column Addition

We will understand this by adding a new column to an existing data frame.

Example

```
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)

# 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([10,20,30],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
```

Its **output** is as follows –

Adding a new column by passing as Series:

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Adding a new column using the existing columns in DataFrame:

	one	two	three	four
a	1.0	1	10.0	11.0
b	2.0	2	20.0	22.0
c	3.0	3	30.0	33.0
d	NaN	4	NaN	NaN

Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

Example

```
# Using the previous DataFrame, we will delete a column  
# using del function  
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']),  
     '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 the first column using DEL function:")
del df['one']
print df

# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print df
```

Its **output** is as follows –

Our dataframe is:

	one	three	two
a	1.0	10.0	1
b	2.0	20.0	2
c	3.0	30.0	3
d	NaN	NaN	4

Deleting the first column using DEL function:

	three	two
a	10.0	1
b	20.0	2
c	30.0	3
d	NaN	4

Deleting another column using POP function:

	three
a	10.0
b	20.0
c	30.0
d	NaN

Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

Selection by Label

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

```
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.loc['b']
```

Its **output** is as follows –

```
one 2.0  
two 2.0  
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

```
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.iloc[2]
```

Its **output** is as follows –

```
one 3.0  
two 3.0  
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ‘ : ’ operator.

```
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[2:4]
```

Its **output** is as follows –

```
   one two
c  3.0   3
d  NaN   4
```

Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)
print df
```

Its **output** is as follows –

```
   a  b
0  1  2
1  3  4
0  5  6
1  7  8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0
df = df.drop(0)

print df
```

Its **output** is as follows –

```
a b
1 3 4
1 7 8
```

In the above example, two rows were dropped because those two contain the same label 0.

Python | Pandas DataFrame

Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the **data**, **rows**, and **columns**.

We will get a brief insight on all these basic operation which can be performed on Pandas DataFrame :

- [Creating a DataFrame](#)
- [Dealing with Rows and Columns](#)
- [Indexing and Selecting Data](#)
- [Working with Missing Data](#)
- [Iterating over rows and columns](#)

Creating a Pandas DataFrame

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc. Dataframe can be created in different ways here are some ways by which we create a dataframe:

Creating a dataframe using List: DataFrame can be created using a single list or a list of lists.

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

# list of strings
lst = ['A', 'B', 'C', 'D',
       'F', 'G', 'H']

# Calling DataFrame constructor on
list
df = pd.DataFrame(lst)
print(df)
```

Output:

Creating DataFrame from dict of ndarray/lists: To create DataFrame from dict of ndarray/list, all the ndarray must be of same length. If index is passed then the length index should be equal to the length of arrays. If no index is passed, then by default, index will be range(n) where n is the array length.


```
# Python code demonstrate creating  
# DataFrame from dict narray / lists  
# By default addresses.
```

```
import pandas as pd
```

```
# initialise data of lists.  
data = {'Name':['Tom', 'nick', 'krish',  
             'jack'],  
        'Age':[20, 21, 19, 18]}
```

```
# Create DataFrame  
df = pd.DataFrame(data)
```

```
# Print the output.  
print(df)
```

Output:

Dealing with Rows and Columns

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. We can perform basic operations on rows/columns like selecting, deleting, adding, and renaming.

Column Selection: In Order to select a column in Pandas DataFrame, we can either access the columns by calling them by their columns name.

```
# Import pandas package  
import pandas as pd
```

```
# Define a dictionary containing employee data  
data = {'Name':['Fruded', 'Molten', 'Nive'],  
        'Age':[27, 24, 20],  
        'Address':['New York', 'Germany', 'India'],  
        'Qualification':['Msc', 'MA', 'Phd']}
```

```
# Convert the dictionary into DataFrame  
df = pd.DataFrame(data)
```

```
# select two columns
print(df[['Name', 'Qualification']])
```

Output:

Row Selection: Pandas provide a unique method to retrieve rows from a Data frame. [DataFrame.loc\[\]](#) method is used to retrieve rows from Pandas DataFrame. Rows can also be selected by passing integer location to an [iloc\[\]](#) function.

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("nba.csv", index_col
="Name")

# retrieving row by loc method
first = data.loc["Avery Bradley"]
second = data.loc["R.J. Hunter"]

print(first, "\n\n", second)
```

Output:

As shown in the output image, two series were returned since there was only one parameter both of the times.

Indexing and Selecting Data

Indexing in pandas means simply selecting particular rows and columns of data from a DataFrame. Indexing could mean selecting all the rows and some of the columns, some of the

rows and all of the columns, or some of each of the rows and columns. Indexing can also be known as **Subset Selection**.

Indexing a DataFrame using indexing operator [] :

Indexing operator is used to refer to the square brackets following an object. The [.loc](#) and [.iloc](#) indexers also use the indexing operator to make selections. In this indexing operator to refer to `df[]`.

Selecting a single columns

In order to select a single column, we simply put the name of the column in-between the brackets

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("nba.csv", index_col
="Name")

# retrieving columns by indexing operator
first = data["Age"]

print(first)
```

Indexing a DataFrame using [.loc\[\]](#) :

This function selects data by the **label** of the rows and columns. The `df.loc` indexer selects data in a different way than just the indexing operator. It can select subsets of rows or columns. It can also simultaneously select subsets of rows and columns.

Selecting a single row

In order to select a single row using `.loc[]`, we put a single row label in a `.loc` function.

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("nba.csv", index_col
="Name")

# retrieving row by loc method
first = data.loc["Avery Bradley"]
second = data.loc["R.J. Hunter"]

print(first, "\n\n", second)
```

Output:

As shown in the output image, two series were returned since there was only one parameter both of the times.

Indexing a DataFrame using [.iloc\[\]](#) :

This function allows us to retrieve rows and columns by position. In order to do that, we'll need to specify the positions of the rows that we want, and the positions of the columns that we want as well. The `df.iloc` indexer is very similar to `df.loc` but only uses integer locations to make its selections.

Selecting a single row

In order to select a single row using `iloc[]`, we can pass a single integer to `.iloc[]` function.

```
import pandas as pd

# making data frame from csv file
data = pd.read_csv("nba.csv", index_col
="Name")

# retrieving rows by iloc method
row2 = data.iloc[3]

print(row2)
```

Working with Missing Data

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 scenario. Missing Data can also refer to as NA(Not Available) values in pandas.

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 function can also be used in Pandas Series in order to find null values in a series.

```
# 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)

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

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)

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

Dropping missing values using dropna() :

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

```

# 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)

df

```

Now we drop rows with at least one Nan value (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()

```

Iterating over rows and columns

Iteration is a general term for taking each item of something, one after another. Pandas DataFrame consists of rows and columns so, in order to iterate over dataframe, we have to iterate a dataframe like a dictionary.

Iterating over rows :

In order to iterate over rows, we can use three function `iteritems()`, `iterrows()`, `itertuples()` . These three function will help in iteration over rows.

```
# 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]}

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

print(df)
```

Now we apply `iterrows()` function in order to get a each element of rows.

```
# 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]}

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

# iterating over rows using iterrows() function
for i, j in df.iterrows():
```



```
print(i, j)
print()
```

Iterating over Columns :

In order to iterate over columns, we need to create a list of dataframe columns and then iterating through that list to pull out the dataframe columns.

```
# 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]}

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

print(df)
```

Now we iterate through columns in order to iterate through columns we first create a list of dataframe columns and then iterate through list.

```
# creating a list of dataframe columns
columns = list(df)
```

```
for i in columns:
```

```
    # printing the third element of the
    column
    print (df[i][2])
```

Output:

DataFrame Methods:

Function	Description
index()	Method returns index (row labels) of the DataFrame
<u>insert()</u>	Method inserts a column into a DataFrame
<u>add()</u>	Method returns addition of dataframe and other, element-wise (binary operator add)
<u>sub()</u>	Method returns subtraction of dataframe and other, element-wise (binary operator sub)
<u>mul()</u>	Method returns multiplication of dataframe and other, element-wise (binary operator mul)
<u>div()</u>	Method returns floating division of dataframe and other, element-wise (binary operator truediv)
unique()	Method extracts the unique values in the dataframe
<u>nunique()</u>	Method returns count of the unique values in the dataframe
value_counts()	Method counts the number of times each unique value occurs within the Series
columns()	Method returns the column labels of the DataFrame

axes()	Method returns a list representing the axes of the DataFrame
<u>isnull()</u>	Method creates a Boolean Series for extracting rows with null values
<u>notnull()</u>	Method creates a Boolean Series for extracting rows with non-null values
between()	Method extracts rows where a column value falls in between a predefined range
<u>isin()</u>	Method extracts rows from a DataFrame where a column value exists in a predefined collection
dtypes()	Method returns a Series with the data type of each column. The result's index is the original DataFrame's columns
<u>astype()</u>	Method converts the data types in a Series
values()	Method returns a Numpy representation of the DataFrame i.e. only the values in the DataFrame will be returned, the axes labels will be removed
sort_values()- <u>Set1</u> , <u>Set2</u>	Method sorts a data frame in Ascending or Descending order of passed Column
<u>sort_index()</u>	Method sorts the values in a DataFrame based on their index positions or labels instead of their values but sometimes a data frame is made out of two or more data frames and hence later index can be changed using this method
<u>loc[]</u>	Method retrieves rows based on index label
<u>iloc[]</u>	Method retrieves rows based on index position
<u>ix[]</u>	Method retrieves DataFrame rows based on either index label or index position. This method combines the best features of the .loc[] and .iloc[] methods
<u>rename()</u>	Method is called on a DataFrame to change the names of the index labels or column names

columns()	Method is an alternative attribute to change the coloumn name
<u>drop()</u>	Method is used to delete rows or columns from a DataFrame
<u>pop()</u>	Method is used to delete rows or columns from a DataFrame
<u>sample()</u>	Method pulls out a random sample of rows or columns from a DataFrame
<u>nsmallest()</u>	Method pulls out the rows with the smallest values in a column
<u>nlargest()</u>	Method pulls out the rows with the largest values in a column
<u>shape()</u>	Method returns a tuple representing the dimensionality of the DataFrame
<u>ndim()</u>	Method returns an 'int' representing the number of axes / array dimensions. Returns 1 if Series, otherwise returns 2 if DataFrame
<u>dropna()</u>	Method allows the user to analyze and drop Rows/Columns with Null values in different ways
<u>fillna()</u>	Method manages and let the user replace NaN values with some value of their own
<u>rank()</u>	Values in a Series can be ranked in order with this method
<u>query()</u>	Method is an alternate string-based syntax for extracting a subset from a DataFrame
<u>copy()</u>	Method creates an independent copy of a pandas object
<u>duplicated()</u>	Method creates a Boolean Series and uses it to extract rows that have duplicate values
<u>drop_duplicates()</u>	Method is an alternative option to identifying duplicate rows and removing them through filtering

<u>set_index()</u>	Method sets the DataFrame index (row labels) using one or more existing columns
<u>reset_index()</u>	Method resets index of a Data Frame. This method sets a list of integer ranging from 0 to length of data as index
<u>where()</u>	Method is used to check a Data Frame for one or more condition and return the result accordingly. By default, the rows not satisfying the condition are filled with NaN value