

# Pandas cheat sheet-Aryan

---

## 1. Introduction to Pandas

---

### What is Pandas?

Pandas is a fast, powerful, flexible, and easy-to-use open-source data analysis and manipulation library for Python. It provides data structures like Series and DataFrames that are designed to handle structured data, offering a variety of tools to clean, manipulate, and analyze large datasets. Pandas is built on top of NumPy, and it integrates well with other data science libraries such as Matplotlib, Seaborn, and SciPy.

The name "Pandas" has a reference to both "[Panel Data](#)", and "[Python Data Analysis](#)" and was created by [Wes McKinney in 2008](#).

### Why Pandas?

- Data set cleaning, merging, and joining.
- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data.
- Columns can be inserted and deleted from DataFrame and higher dimensional objects.
- Powerful group by functionality for performing split-apply-combine operations on data sets.
- Data Visualization.

### Key Features of Pandas

- **Data Cleaning & Preparation:** Efficient handling of missing or corrupted data.
  - **Handling Missing Data:** Includes functions for filling, replacing, and dropping missing data.
  - **Merging & Joining:** Combines multiple datasets in different ways.
  - **Powerful Data Manipulation:** With operations such as slicing, filtering, and transforming data.
  - **Fast Performance:** Optimized for large datasets.
  - **Integrated Visualization:** Direct plotting and integration with Matplotlib.
  - **Works with Different Data Formats:** Supports reading and writing from CSV, Excel, SQL, JSON, etc.
-

## 2. Installation and Setup

---

### Installing Pandas

Python Pandas can be installed on [Windows](#) in two ways:

- Using pip
- Using Anaconda

#### Install Pandas using [pip](#) :

##### Step 1 : Launch Command Prompt

To open the Start menu, press the Windows key or click the Start button. To access the Command Prompt, type “cmd” in the search bar, click the displayed app, or use Windows key + r, enter “cmd,” and press Enter.

##### Step 2 : Run the Command

Pandas can be installed using PIP by use of the following command in Command Prompt.  
pip install pandas

#### Install Pandas using [Anaconda](#)

1. Create a new Jupyter Notebook.
2. Write !pip install pandas to install the Pandas.

#### Install Pandas on [Linux](#)

Install Pandas on Linux, just type the following command in the Terminal Window and press Enter. Linux will automatically download and install the packages and files required to run Pandas Environment in Python:

```
pip3 install pandas
```

#### Install Pandas on [MacOS](#)

Install Pandas on MacOS, type the following command in the Terminal, and make sure that python is already installed in your system.

```
pip install pandas
```

---

#### Importing Pandas:

To import the pandas in the working environment run the following command:  
import pandas as pd

---

## 3. Fundamental Pandas Objects

---

## Objects in Pandas:

Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. fundamental Pandas data structures:

1. Series.
2. DataFrame.

### 3.1 Pandas Series

A **Series** is a one-dimensional labeled array that can hold any data type (integers, floats, strings, etc.). It can be thought of as a column in a table or a simple list with additional features.

#### Creating Series

**Creating an empty Series:** Series() function of Pandas is used to create a series. A basic series, which can be created is an Empty Series.

```
import pandas as pd # import pandas as pd
ser = pd.Series() # Creating empty series
print(ser)
```

**From a List :** In order to create a series from list, we have to first create a list after that we can create a series from list.

```
import pandas as pd
# Creating a Series from a simple list
data = ['a', 'j', 'a', 'y', 's']
ser = pd.Series(data)
print(ser)
```

**From a NumPy Array :** In order to create a series from array, we have to import a numpy module and have to use array() function.

```
import pandas as pd # import pandas as pd
import numpy as np # import numpy as np
data = np.array(['a','j','a','y','s']) # simple array
ser = pd.Series(data)
print(ser)
```

Creating a series from array with an index: In order to create a series by explicitly providing index instead of the default, we have to provide a list of elements to the index parameter with the same number of elements as it is an array.

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

```
import numpy as np # import numpy as np
data = np.array(['g', 'e', 'e', 'k', 's']) # simple array
ser = pd.Series(data, index=[10, 11, 12, 13, 14]) # providing an index
print(ser)
```

### From a Dictionary

```
# Creating a Series from a dictionary
dict_data = {
    'rent': 10,
    'ajay': 20,
    'saurav': 30
}
ser = pd.Series(dict_data)
print(ser)
```

## Accessing Series Elements

An element in the series can be accessed similarly to that in an ndarray. Elements of a series can be accessed in two ways:

- Accessing Element from Series with Position
- Accessing Element Using Label (index)

### By Position

The first five elements of the series are accessed and printed using `print(ser[:5])`.

```
# Accessing the first 5 elements
print(ser[:5])
#Accessing Last 10 Elements of Series
print(ser[-10:])
```

### Accessing First 5 Elements of Series in nba.csv File

```
# importing pandas module
import pandas as pd
# making data frame
df = pd.read_csv("nba.csv")
ser = pd.Series(df['Name'])
ser.head(10)
```

### By Label

In order to access the series element refers to the index number. Use the index [operator \[\]](#) to access an element in a series. The index must be an integer.

```

# Accessing an element by index label
print(ser[10]) # Using custom index
# accessing a multiple element using
# index element
print(ser[[10, 11, 12, 13, 14]])

```

## 4. Series Operations and Functions

### 4.1 Aggregate Functions

An aggregate is a function where we can apply multiple functions at the same time. Advantage of using aggregate function is it allows to apply mutiple functions to a series simultaneously. Series.agg() function is used to apply some aggregation across the series . Aggregate using callable, string, dict, or list of string/callables. Most frequently used aggregations are:

**sum:** Return the sum of the values for the series

**min:** Return the minimum of the values for the series

**max:** Return the maximum of the values for the series.

```

# Multiple aggregation functions at once
sr = pd.Series([1, 2, 3, 4, 5])
result = sr.agg([min, max, sum])
print(result)

```

### 4.2 Common Series Methods

**Absolute Values :** Pandas Series.abs() method is used to get the absolute numeric value of each element in Series.

```

lst = [2, -10.87, -3.14, 0.12]
ser = pd.Series(lst)
abs_ser = ser.abs() # Converts negative values to positive
print(abs_ser)

```

**Appending Series :** Series.append() function is used to concatenate two or more series object.

```

sr1 = pd.Series([3, 5, 7, 1, 9])
sr2 = pd.Series([-1, -4, -7, -10])

```

```
sr_combined = sr1.append(sr2)
print(sr_combined)
```

**Astype Function :** Series.astype() function is used to convert from one data type to another.

```
sr = pd.Series([2, 9, 1, 5, 8])
float_ser = sr.astype('float') # Converting to float type
print(float_ser)
```

**Between Function :** Pandas between() method is used on series to check which values lie between first and second argument.

```
series = pd.Series([10, 45, 23, 27, 98, 86])
filtered_series = series.between(0, 30) # Filter values between 0 and 30
print(filtered_series)
```

**Map Function :** The main task of map() is used to map the values from two series that have a common column. To map the two Series, the last column of the first Series should be the same as the index column of the second series, and the values should be unique.

```
import pandas as pd
import numpy as np
a = pd.Series(['Java', 'C', 'C++', np.nan])
a.map({'Java': 'Core'})
a.map('I like {}'.format, na_action='ignore')
```

**Std Function :** The Pandas std() is defined as a function for calculating the standard deviation of the given set of numbers, DataFrame, column, and rows. In respect to calculate the standard deviation, we need to import the package named "statistics" for the calculation of median.

```
import pandas as pd
# calculate standard deviation
import numpy as np
print(np.std([4,7,2,1,6,3]))
print(np.std([6,9,15,2,-17,15,4]))
```

**Pandas Series.to\_frame() :** Series is defined as a type of list that can hold an integer, string, double values, etc. It returns an object in the form of a list that has an index starting from 0 to n where n represents the length of values in Series.

The main difference between Series and Data Frame is that Series can only contain a single list with a particular index, whereas the DataFrame is a combination of more than one series that can analyze the data.

The Pandas Series.to\_frame() function is used to convert the series object to the DataFrame.

```
s = pd.Series(["a", "b", "c"],  
name="vals")  
s.to_frame()
```

**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. <b>index</b>	Defines the index of the Series.
Series. <b>shape</b>	It returns a tuple of shape of the data.
Series. <b>dtype</b>	It returns the data type of the data.
Series. <b>size</b>	It returns the size of the data.
Series. <b>empty</b>	It returns True if Series object is empty, otherwise returns false.
Series. <b>hasnans</b>	It returns True if there are any NaN values, otherwise returns false.
Series. <b>nbytes</b>	It returns the number of bytes in the data.
Series. <b>ndim</b>	It returns the number of dimensions in the data.
Series. <b>itemsizes</b>	It returns the size of the datatype of item.

## 5. DataFrame - Advanced Data Structures

### 5.1 What is a DataFrame?

A **DataFrame** is a two-dimensional labeled data structure with columns of potentially different types. It's essentially a table with rows and columns, much like a spreadsheet or SQL table. A DataFrame is the most important and versatile object in Pandas.

### Parameter & Description:

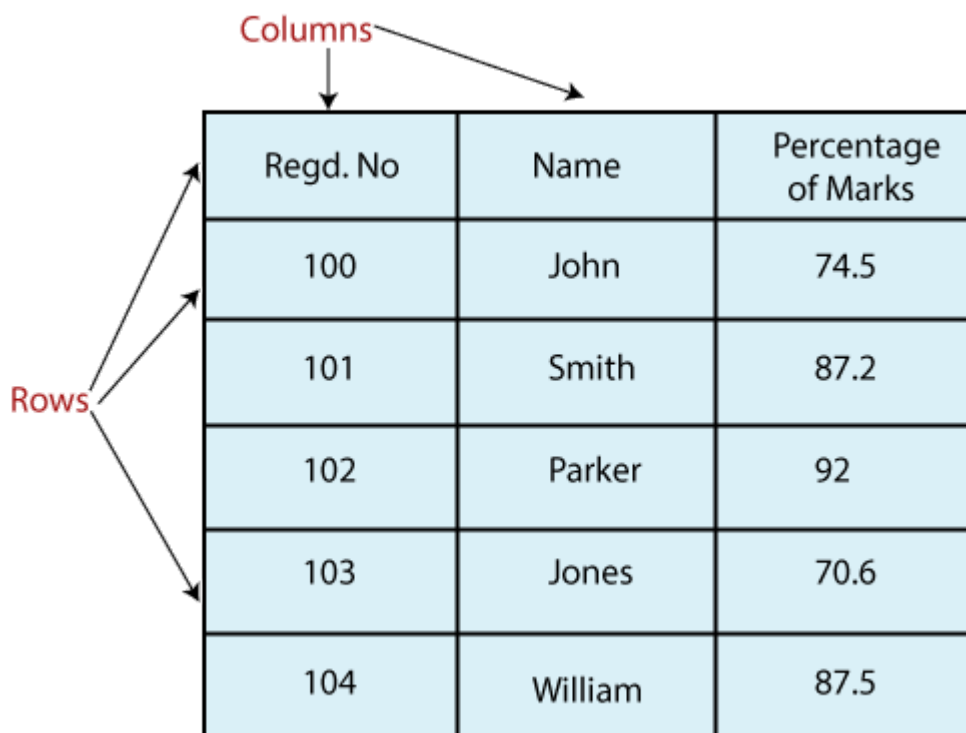
**data:** It consists of different forms like ndarray, series, map, constants, lists, array.

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

**columns:** The default syntax is np.arange(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.



The diagram shows a table with 6 rows and 3 columns. The columns are labeled 'Regd. No', 'Name', and 'Percentage of Marks'. The rows are labeled with registration numbers 100, 101, 102, 103, 104. Arrows point from the labels 'Columns' and 'Rows' to their respective parts of the table.

Regd. No	Name	Percentage of Marks
100	John	74.5
101	Smith	87.2
102	Parker	92
103	Jones	70.6
104	William	87.5

### Creating DataFrames

**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)
```

#### From a 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)
```

#### From a Dictionary

```
# Creating a DataFrame from a dictionary
data = {'Name': ['Tom', 'Nick', 'John'],
        'Age': [20, 21, 19]}
df = pd.DataFrame(data)
print(df)
```

#### From a CSV File

```
# Loading a DataFrame from a CSV file
df = pd.read_csv('data.csv')
print(df.head()) # Display first 5 rows
```

## 5.2 DataFrame Operations

### Indexing and Slicing

```
# Selecting specific columns
df['Name'] # Select 'Name' column
df[['Name', 'Age']] # Select multiple columns
```

### Adding New Columns

```
# Adding a new column
df['Country'] = ['USA', 'Canada', 'UK']
```

### Columns Deleting :

```
# importing the pandas library
import pandas as pd
info = {'one' : pd.Series([1, 2], index= ['a', 'b']),
```

```

    'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}
df = pd.DataFrame(info)
print ("The DataFrame:")
print (df)
# using del function
print ("Delete the first column:")
del df['one']
print (df)
# using pop function
print ("Delete the another column:")
df.pop('two')
print (df)

```

## Handling Missing Data

```

# Filling missing values with a default value
df.fillna(0)

# Dropping rows with missing data
df.dropna()

```

## DataFrame Functions

There are lots of functions used in DataFrame which are as follows:

Functions	Description
<a href="#">Pandas DataFrame.append()</a>	Add the rows of other dataframe to the end of the given dataframe.
<a href="#">Pandas DataFrame.apply()</a>	Allows the user to pass a function and apply it to every single value of the Pandas series.
<a href="#">Pandas DataFrame.assign()</a>	Add new column into a dataframe.
<a href="#">Pandas DataFrame.astype()</a>	Cast the Pandas object to a specified dtype.astype() function.
<a href="#">Pandas DataFrame.concat()</a>	Perform concatenation operation along an axis in the DataFrame.

<a href="#"><u>Pandas DataFrame.count()</u></a>	Count the number of non-NA cells for each column or row.
<a href="#"><u>Pandas DataFrame.describe()</u></a>	Calculate some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame.
<a href="#"><u>Pandas DataFrame.drop_duplicates()</u></a>	Remove duplicate values from the DataFrame.
<a href="#"><u>Pandas DataFrame.groupby()</u></a>	Split the data into various groups.
<a href="#"><u>Pandas DataFrame.head()</u></a>	Returns the first n rows for the object based on position.
<a href="#"><u>Pandas DataFrame.hist()</u></a>	Divide the values within a numerical variable into "bins".
<a href="#"><u>Pandas DataFrame.iterrows()</u></a>	Iterate over the rows as (index, series) pairs.
<a href="#"><u>Pandas DataFrame.mean()</u></a>	Return the mean of the values for the requested axis.
<a href="#"><u>Pandas DataFrame.melt()</u></a>	Unpivots the DataFrame from a wide format to a long format.
<a href="#"><u>Pandas DataFrame.merge()</u></a>	Merge the two datasets together into one.
<a href="#"><u>Pandas DataFrame.pivot_table()</u></a>	Aggregate data with calculations such as Sum, Count, Average, Max, and Min.
<a href="#"><u>Pandas DataFrame.query()</u></a>	Filter the dataframe.
<a href="#"><u>Pandas DataFrame.sample()</u></a>	Select the rows and columns from the dataframe randomly.
<a href="#"><u>Pandas DataFrame.shift()</u></a>	Shift column or subtract the column value with the previous row value from the dataframe.
<a href="#"><u>Pandas DataFrame.sort()</u></a>	Sort the dataframe.
<a href="#"><u>Pandas DataFrame.sum()</u></a>	Return the sum of the values for the requested axis by the user.
<a href="#"><u>Pandas DataFrame.to_excel()</u></a>	Export the dataframe to the excel file.
<a href="#"><u>Pandas DataFrame.transpose()</u></a>	Transpose the index and columns of the dataframe.

<a href="#">Pandas DataFrame.where()</a>	Check the dataframe for one or more conditions.
--	---

## 6. Working with SQL Databases using Pandas

Pandas provides powerful tools for interacting with SQL databases, making it easy to read, write, and manipulate data stored in relational databases.

### SQL Database Connectivity:

```
import pandas as pd
import sqlite3 # Or other database connectors like psycopg2 for PostgreSQL,
               pymysql for MySQL

# Establishing a connection to a SQLite database
conn = sqlite3.connect('example_database.db')

# Read SQL query or database table into a DataFrame
df = pd.read_sql_query("SELECT * FROM customers", conn)

# Write DataFrame to SQL database
df.to_sql('table_name', conn, if_exists='replace')

# Close the connection
conn.close()
```

### Key Methods for SQL Interaction:

1. `pd.read_sql_query()`: Execute a SQL query and load results into a DataFrame
2. `pd.read_sql_table()`: Read entire table from database
3. `DataFrame.to_sql()`: Write DataFrame to SQL database

### Common SQL-like Operations with Pandas:

```
# Equivalent to SQL WHERE clause
```

```
filtered_df = df[df['column_name'] > value]
```

```
# Equivalent to SQL GROUP BY
```

```
grouped_df = df.groupby('category')['sales'].sum()
```

```
# Equivalent to SQL JOIN
```

```
merged_df = pd.merge(df1, df2, on='common_column')
```

## 7. Data Visualization with Matplotlib and Pandas

Pandas integrates seamlessly with Matplotlib to create various types of visualizations directly from DataFrames.

### Basic Plotting:

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

```
# Line Plot
```

```
df['column'].plot(kind='line')
```

```
plt.title('Line Plot')
```

```
plt.show()
```

```
# Bar Plot
```

```
df.plot(kind='bar')
```

```
plt.title('Bar Plot')
```

```
plt.show()
```

```
# Histogram
```

```
df['column'].plot(kind='hist', bins=20)
```

```
plt.title('Histogram')
```

```
plt.show()
```

```
# Scatter Plot
```

```
df.plot(kind='scatter', x='column1', y='column2')
```

```
plt.title('Scatter Plot')
```

```
plt.show()
```

### Advanced Visualization Techniques:

```
# Multiple plots in one figure
```

```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
```

```

df['column1'].plot(kind='line', ax=axes[0, 0])
df['column2'].plot(kind='bar', ax=axes[0, 1])
df.boxplot(ax=axes[1, 0])
df.plot(kind='area', ax=axes[1, 1])
plt.tight_layout()
plt.show()

```

### # Customizing Plots

```

df['column'].plot(
    kind='line',
    color='red',
    linewidth=2,
    marker='o',
    title='Customized Line Plot'
)
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.show()

```

Descriptive Plotting:

```

# Quick statistical visualizations
df.plot(kind='box') # Box plot for distribution
df.plot(kind='density') # Density plot
df.hist() # Histogram for each numeric column

```

These sections provide an introduction to working with SQL databases and creating visualizations using Pandas and Matplotlib. They demonstrate how to:

- Connect to databases
- Execute SQL-like operations
- Create various types of plots
- Customize visualizations
- Perform quick statistical visualizations

## 8. Best Practices and Tips

- Import pandas as **pd** for consistency and brevity.

- **Use appropriate data types:** Explicitly cast data when necessary.
- **Handle missing data:** Utilize `fillna()`, `dropna()`, or forward/backward fill.
- **Use vectorized operations:** Pandas operations are typically faster than iterating through rows.
- **Leverage built-in functions** for common tasks like aggregation and transformation.

## 9. Next Steps

- **Explore DataFrame operations:** Learn more about filtering, grouping, and summarizing data.
- **Data cleaning techniques:** Focus on removing duplicates, handling missing values, and correcting data types.
- **Advanced data manipulation:** Master functions like `groupby()`, `pivot_table()`, and `merge()`.
- **Data visualization:** Learn how to visualize data with Pandas and integrate it with libraries like Matplotlib.
- **Work on real-world datasets:** Practice by analyzing datasets from sources like Kaggle, UCI, or government databases.

## 10. Recommended Resources

- **Official Pandas Documentation:** [Pandas Docs](#)
- **Python for Data Analysis** (Book by Wes McKinney): Comprehensive guide to Pandas.
- **Kaggle Datasets and Tutorials:** Practice with real-world datasets.
- **DataCamp Pandas Courses:** Learn Pandas in an interactive environment.