

<https://pandas.pydata.org/docs/>

<https://github.com/pandas-dev/pandas>

<https://realpython.com/pandas-dataframe/>

**1. Foundation: Essential Python Concepts**

Before diving into Pandas, we need to understand some Python fundamentals:

* LIST [ ]: list is a versatile data structure used to store a collection of items.

Example:

my\_list = [1, 2, 3, "hello", True]

* **Dictionaries**: We must understand how to create and manipulate dictionaries, as they are pivotal in constructing DataFrames. stores data as key-value pairs, allowing you to access a specific value quickly by using its unique "key" to look it up, similar to how a traditional dictionary works with words and definitions; it is created using curly braces {} and is considered an unordered collection of data.

Example:

Key: Value pair

data = {

'Name': ['Dhairya', 'Patrick', 'Arshad'],

'Age': [22, 21, 16],

'City': ['Ahmedabad', 'Manila', 'Pune']

}

data['Name']

* **List Comprehensions**: Concept of list comprehensions is very important for efficient data processing.

*Example:*

squares = [x\*\*2 for x in range(10)]

**Equivalent Java code for above 1 line:**

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Integer> squares = new ArrayList<>();

for (int x = 0; x < 10; x++) {

squares.add(x \* x);

}

System.out.println(squares);

}

}

* **Comprehensions**: Familiarize yourself with list and dictionary comprehensions for concise data manipulation.

Example:

square\_dict = {x: x\*\*2 for x in range(5)}

* **Slice Operator**: We can easily extract subsets from lists or strings using slicing.

Example:

numbers = [0, 1, 2, 3, 4, 5]

subset = numbers[1:4]

Output: [1, 2, 3]

* **Lambda Functions:** A small, anonymous function defined using the lambda keyword, used for short, simple operations that are not reused elsewhere.

**Format:**

***lambda arguments: expression***

*Example:*

add\_10 = lambda x: x + 10

print(add\_10(5))

# Output: 15

**2. Introduction to Pandas: Series and DataFrames**

With the basics in place, delve into Pandas' core data structures:

* **What is a Series?**

*A one-dimensional labeled array capable of holding any data type.*

Example:

import pandas as pd

ages = pd.Series([22, 21, 16])

print(ages)

*0 25*

*1 30*

*2 35*

*dtype: int64*

ages = pd.Series([25, 30, 35], index=['Dhairya', 'Patrick', 'Arshad'])

print(ages)

*Dhairya 25*

*Patrick 30*

*Arshad 35*

*dtype: int64*

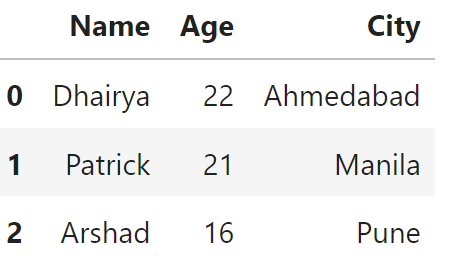
* **DataFrames**: A two-dimensional labeled data structure, akin to a table in SQL or an Excel spreadsheet.

Example:

# Creating a DataFrame from our previously created dictionary

df = pd.DataFrame(data)

df



**There are tons of ways we can create a DataFrame.**

**1. We saw creation using a dictionary, which is the most common.**

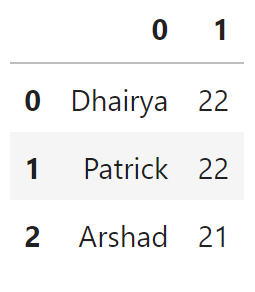
**2. From a List of Lists**

We can pass a list of lists and specify column names:

data = [['Dhairya', 22], ['Patrick', 22], ['Arshad', 21]]

df = pd.DataFrame(data)

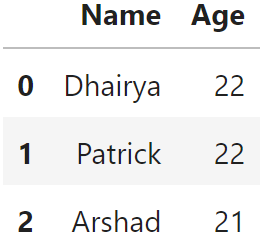
print(df)



#Above you will see the column names are 0, and 1, which doesn’t look good, we need to have some good names, for the columns

We can do that by passing param “column”:

pd.DataFrame(data, columns=['Name','Age'])



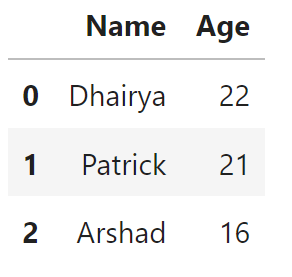
**3. From a List of Dictionaries**

Each dictionary represents a row, and the keys are column names:

data = [{'Name': 'Dhairya', 'Age': 22}, {'Name': 'Patrick', 'Age': 21}, {'Name': 'Arshad', 'Age': 16}]

df = pd.DataFrame(data)

print(df)

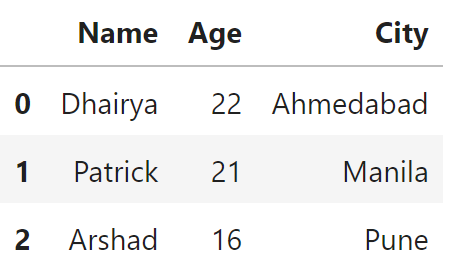


**4. From a CSV File**

We can load data from a CSV file:

df = pd.read\_csv('sample\_data.csv')

df

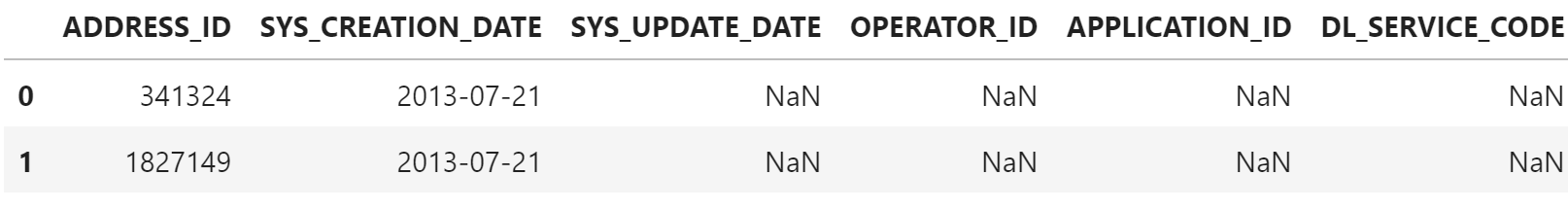


**5. From an Excel File**

df = pd.read\_excel('SAMPLE\_ADDRESS.xlsx') 

df

//We can also add a param, sheet\_name=’’, as an excel can have multiple excel sheets, to have the specific sheet loaded in dataframe.



**6. From a Dictionary with Series**

Each column can be a Pandas Series:

data = {

'Name': pd.Series(['Dhairya', 'Patrick', 'Arshad']),

'Age': pd.Series([22, 21, 16]),

'City': pd.Series(['Ahmedabad', 'Manila', 'Pune'])

}

df = pd.DataFrame(data)

print(df)

**7. Using read\_sql()**

Load data directly from a SQL query:

import sqlite3

conn = sqlite3.connect(':memory:')

conn.execute("CREATE TABLE data (Name TEXT, Age INTEGER, City TEXT)")

conn.execute("INSERT INTO data (Name, Age, City) VALUES ('Dhairya', 22, 'Ahmedabad'), ('Patrick', 21, 'Manila'), ('Arshad', 16, 'Pune')")

df = pd.read\_sql("SELECT \* FROM data", conn)

df

**3. Let’s create with the same data as we had above: Login to any environment, you can create below table.**

CREATE TABLE Persons (

Name VARCHAR2(50),

Age NUMBER(3),

City VARCHAR2(50)

);

INSERT INTO Persons (Name, Age, City) VALUES ('Dhairya', 22, 'Ahmedabad');

INSERT INTO Persons (Name, Age, City) VALUES ('Patrick', 21, 'Manila');

INSERT INTO Persons (Name, Age, City) VALUES ('Arshad', 16, 'Pune');

|  |  |  |
| --- | --- | --- |
| SQL QUERY | Pandas Code | Details |
| SELECT Name, Age FROM Persons; | df[['Name', 'Age']] | **SELECT Equivalent**: Use DataFrame indexing to select columns. |
| desc Persons | df.columns | Referencing the **columns** attribute of the DataFrame. It returns an **Index** object that contains the column labels of the DataFrame. |
| SELECT \* FROM Persons WHERE Age > 30; | df['Age'] > 20   * This command will return a Series of Bool values, either True or False, when the same is passed as a param to the parent DataFrame, it will return all rows which has true index value.     df[df[‘Age’ > 20]] | **WHERE Clause**: Apply boolean indexing to filter rows. |
| SELECT City, COUNT(\*) FROM Persons GROUP BY City; | df.groupby('City').size()    df.groupby('City')['Age'].mean() | **GROUP BY and Aggregations**: Utilize groupby() for grouping and aggregation functions like sum(), mean(). |
| SELECT CAST('123.45' AS NUMBER) FROM dual; | df.to\_numeric(df[‘Age’], errors=’coerce’)   * Converts to float by default.   df[‘Age’]= df[‘Age’].astype(float)   * This has limited handling options, will raise an error if conversion fails, also cannot handle non-numeric data, will handle if the values are already numeric(float, int, double). * Converts to int64 by default. | **TypeCasting:**  **Param errors:**   * If errors='raise' (default): Raises an error if any non-numeric values are found. * if errors='coerce': Converts non-numeric values to NaN (useful for cleaning data). * if errors='ignore': Leaves non-numeric values unchanged. |
| SELECT \* FROM df1 JOIN df2 ON df1.key = df2.key; | pd.merge(df1, df2, on='key') | **JOIN Operations**: Using merge() to combine DataFrames. |
| SELECT \* FROM df1 LEFT JOIN df2 on 1.key = df2.key  Where df2.column is null; | merged\_left = pd.merge(df1, df2, on='ID', how='left') |  |
| desc Persons; | df.describe()    df.info(); | **Describe Operation: Give an overview of what we numerical columns data.** |
| Select \* from Persons where Age=22 and Name=’Dhairya’;  We cannot manipulate at index level in Oracle but Pandas we can. | df.loc[row\_name, column\_name];    Index based:  df.iloc[row\_index, column\_index] | **Fetching specific data from DataFrame** |
| UPDATE Persons set Name=’Pratik’ Where Name=’Patrick’; | df.at[1,’Name’]= ‘Pratik’ | **Updating specific Value in DataFrame** |
| ALTER TABLE Persons ADD Cur\_location varchar(100);  UPDATE Persons SET Cur\_location= ‘Chicago’; | df['Cur\_location']='Chicago' | **Adding new Column and Populating column values** |
| ALTER TABLE Persons DROP COLUMN Cur\_location; | df.drop(‘Cur\_location', axis=1, inplace=True)   * Remember it requires 3 important params: * Column\_name * axis: we need to drop row or column: if row=0, if column =1 * inplace=True , means we want the changes to be committed. | **Dropping a column** |

**Applying Arithmetic Operations**: Perform element-wise operations and utilize functions from NumPy and SciPy.

|  |  |  |
| --- | --- | --- |
| UPDATE Persons set Age= Age + 5; | df[‘Age’] = df[‘Age’] + 1 | **Arithmetic Operations** |
| ALTER TABLE Persons ADD Age\_sqrt NUMBER;  UPDATE Persons SET Age\_sqrt = SQRT(Age); | import numpy as np  df['Age\_Sqrt'] = np.sqrt(df['Age']) | **Operations like Square Root over all columns** |
| Select \* from Persons Order by Age Asc; | df.sort\_values(by='Age', ascending=False).reset\_index() | **Sort DataFrames by index or by column values.** |

### ****Handling Missing Data (One of the Most Important concepts)****

Missing data will be a common challenge while working with real-world datasets. Pandas provides several methods to detect and handle missing values:

#### **1. Detecting Missing Data**

Use isnull() and notnull() to identify missing values in a DataFrame or Series.

**Example:**

data = {

'Name': ['Dhairya', 'Patrick', 'Arshad', None],

'Age': [22, np.nan, 16, 40],

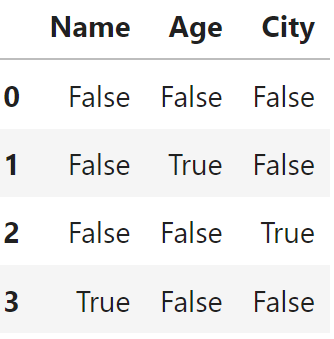
'City': ['Ahmedabad', 'Manila', None, 'Chicago']

}

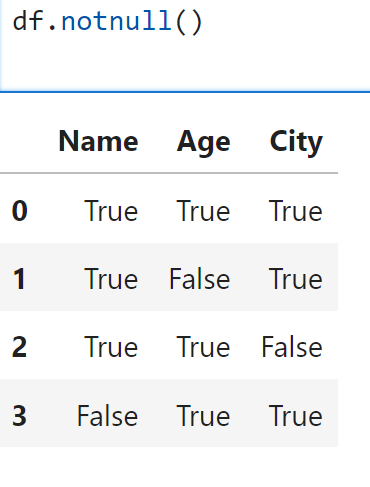
df = pd.DataFrame(data)

# Detect missing values

df.isnull()

****

**Same can be done using notnull()**

****

* **Now let’s see how handy these come in identify column specific data.**
* print(df['City'].isnull().sum())

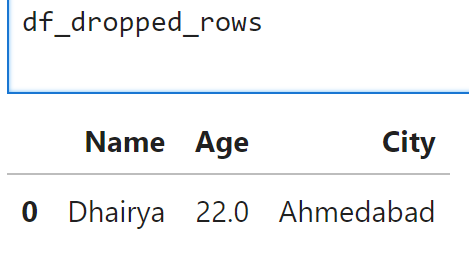
#### **2. Dropping Missing Data**

Use dropna() to remove rows or columns containing missing values.

**Example:**

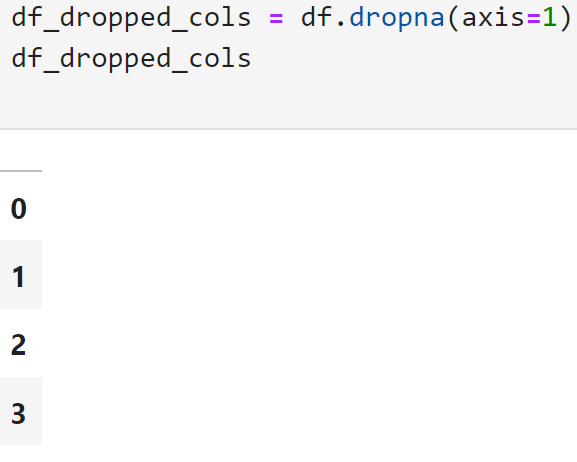
# Drop rows with any missing values

df\_dropped\_rows = df.dropna()



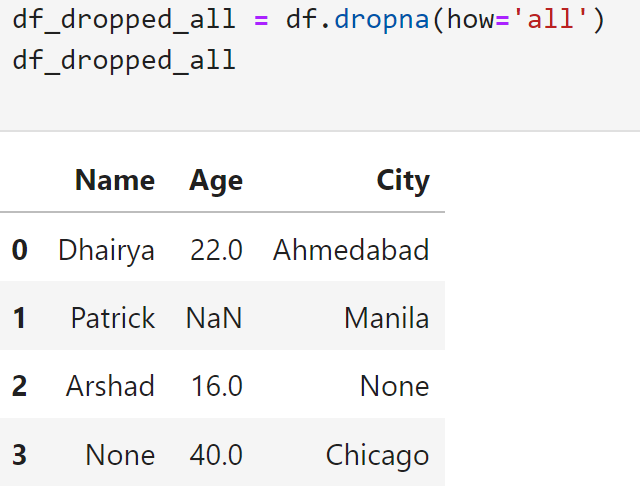
# Drop columns with any missing values

df\_dropped\_cols = df.dropna(axis=1)



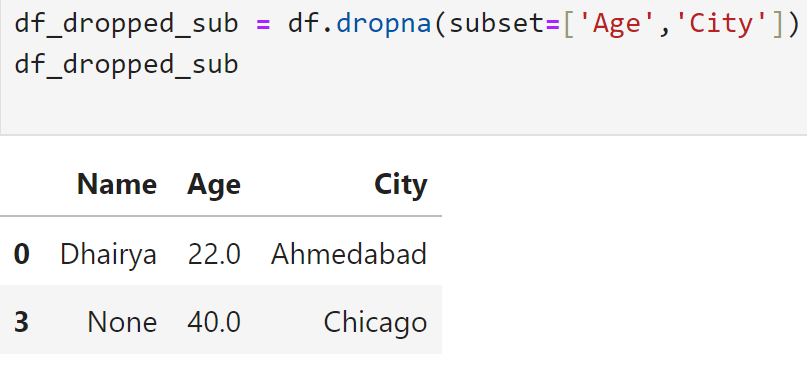
# Drop rows only if all values are missing

df\_dropped\_all = df.dropna(how='all')



# Drop rows where specific columns have missing values

df\_dropped\_subset = df.dropna(subset=['Age', 'City'])

****

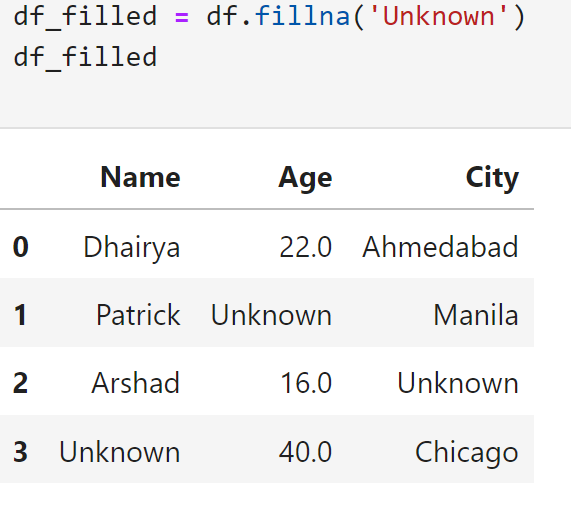
#### **3. Filling Missing Data**

We will be using fillna() to replace missing values with specific values or computed statistics in our datasets.

**Example:**

# Fill missing values with a specific value

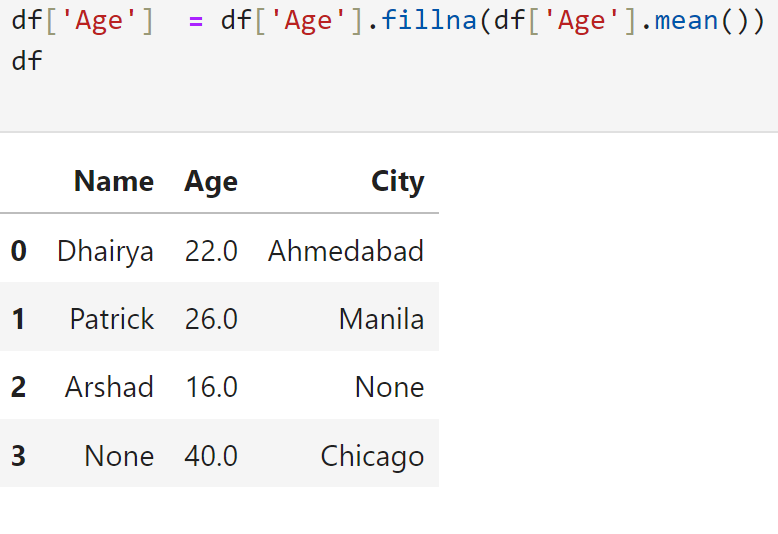
df\_filled = df.fillna('Unknown')



* this is not a good way to handle missing values, as Age is a numeric column, and populating a String would create problems in future.

# Right way of handling Age column

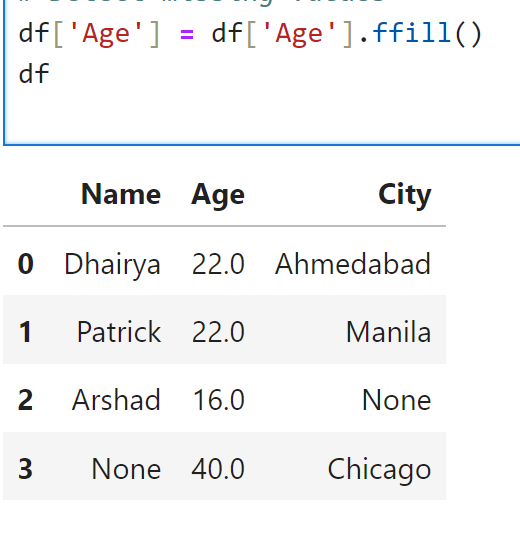
df['Age'] = df['Age'].fillna(df['Age'].mean())



# Fill forward (propagate last valid value forward)

df\_ffill = df.fillna(method='ffill') -- Old

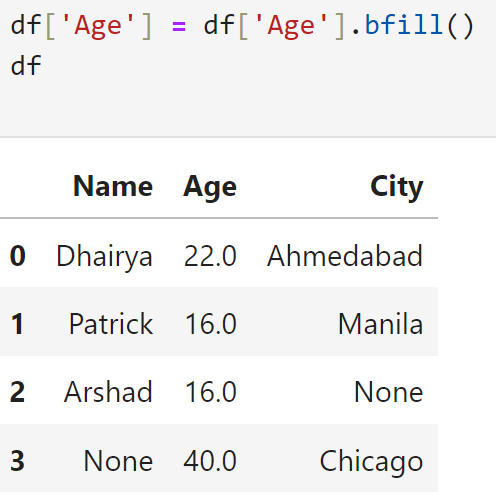
df['Age'] = df['Age'].ffill() – Latest Version



# Fill backward (propagate next valid value backward)

df\_bfill = df.fillna(method='bfill')

df['Age'] = df['Age'].bfill()

****

#### **Merging and Joining DataFrames**

Combine datasets using methods analogous to SQL joins.

**Example:**

data = {

'Name': ['Dhairya', 'Patrick', 'Arshad', 'Rakesh'],

'Age': [22, np.nan, 16, 24],

'City': ['Ahmedabad', 'Manila', None, 'Chicago']

}

df1=pd.DataFrame(data)

sports={

'Name': ['Dhairya', 'Patrick', 'Arshad', 'Nirav'],

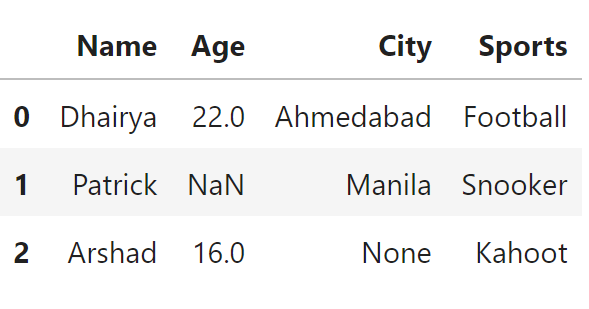
'Sports': ['Football', 'Snooker', 'Kahoot' , 'BodyBuilding' ]

}

df2=pd.DataFrame(sports)

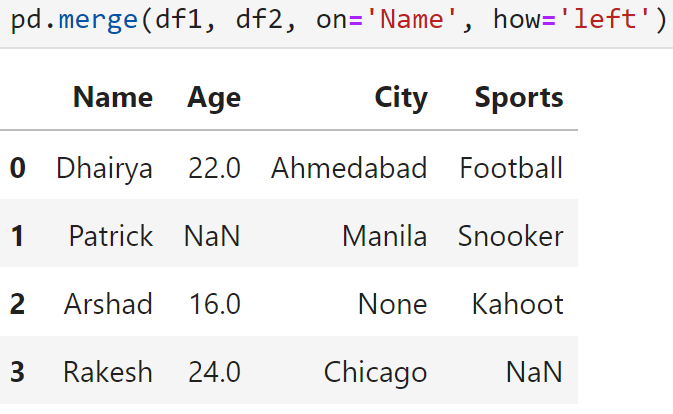
**# Inner Join (default)**

pd.merge(df1, df2, on='Name')

****

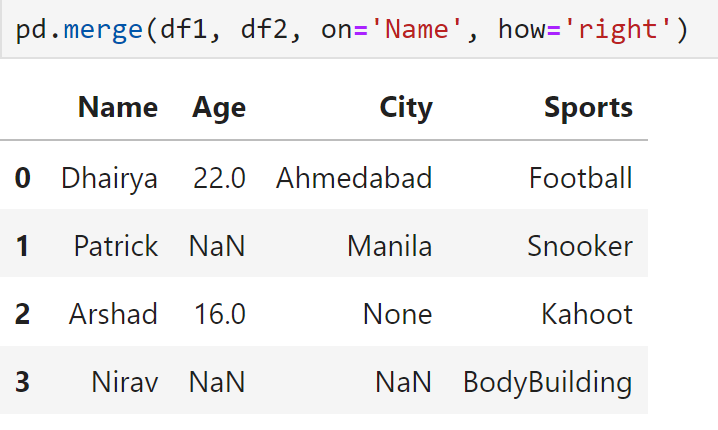
# Left Join

merged\_left = pd.merge(df1, df2, on='ID', how='left')



# Outer Join

merged\_outer = pd.merge(df1, df2, on='ID', how='outer')

****

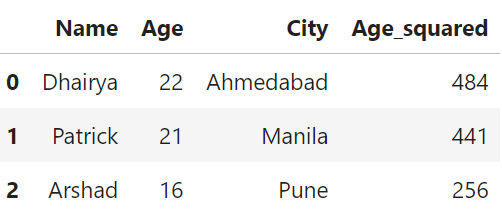
**Some Important functions:**

apply() function:

The apply() function in pandas is a versatile method used to apply custom functions to either rows or columns in a DataFrame. When combined with lambda, it provides a concise way to perform complex operations on data.

df['Age\_squared'] = df['Age'].apply(lambda x: x \*\* 2)

df

****

**Part of Defect Management Code:**

# reading the Excel with defect list

Defect\_file\_path='C:/Users/arshaikh/Udacity DATA Science/Defect\_management project/Sample\_defects\_data.xlsx'

df = pd.read\_excel( Defect\_file\_path, dtype=str)

#reading the HF file

hf\_file\_path = "C:/Users/arshaikh/Udacity DATA Science/Defect\_management project/PythonApplication1/Final\_HF\_List.xlsx"

hf\_df = pd.read\_excel(hf\_file\_path, sheet\_name="Sheet1")

# We can Update the sheet\_name if you have multiple sheets

#.str.strip() method is applied to each element (column name) in the df.columns Index removing leading and trailing whitespaces

#.str.lower() method converts each string (column name) in the Index to **lowercase**

hf\_df.columns = hf\_df.columns.str.strip().str.lower()

#errors=’coerce’ param replaces all value which could not get converted into Number with NaN

hf\_df['hf number'] = pd.to\_numeric(hf\_df['hf number'], errors='coerce')

#dropping all rows which does not contain HF number

hf\_df = hf\_df.dropna(subset=['hf number'])

#first to\_numeric 🡪 astype

hf\_df['hf number'] = hf\_df['hf number'].astype(int)

#same as we did for hf\_df

df.columns = df.columns.str.strip()

# Converting here the 'hot fix id' column in the uploaded df to numeric to be safe and then convert to int same as we did for hf\_df

df['Hot Fix ID'] = pd.to\_numeric(df['Hot Fix ID'], errors='coerce').fillna(0).astype(int)

#Performing the final merge based on 'hot fix id' from df and 'hf number' from hf\_df

df = pd.merge(df, hf\_df[['hf number', 'file name', 'bounce']], how='left', left\_on='Hot Fix ID', right\_on='hf number')

df.head(1)