In [1]:
```
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Alice", "David"],
    "Age": [25, 30, 25, 40],
    "City": ["NY", "LA", "NY", "Chicago"]
}

df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

df_cleaned = df.drop_duplicates()

print("\nModified DataFrame (no duplicates)")
print(df_cleaned)
```

```
Original DataFrame:
    Name  Age     City
0  Alice   25       NY
1    Bob   30       LA
2  Alice   25       NY
3  David   40  Chicago

Modified DataFrame (no duplicates)
    Name  Age     City
0  Alice   25       NY
1    Bob   30       LA
3  David   40  Chicago
```

Syntax:

DataFrame.drop_duplicates(subset=None, keep='first', inplace=False)

Parameters:

1. subset: Specifies the columns to check for duplicates. If not provided all columns are considered.

2. keep: Finds which duplicate to keep:

'first' (default): Keeps the first occurrence, removes subsequent duplicates.

'last': Keeps the last occurrence and removes previous duplicates.

False: Removes all occurrences of duplicates.

3. inplace: If True it modifies the original DataFrame directly. If False (default), returns a new DataFrame.

Return type: Method returns a new DataFrame with duplicates removed unless inplace=True.

In [2]:
```
#1. Dropping Duplicates Based on Specific Columns
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'David'],
    'Age': [25, 30, 25, 40],
    'City': ['NY', 'LA', 'SF', 'Chicago']
})

df_cleaned = df.drop_duplicates(subset=["Name"])

print(df_cleaned)
```

```
    Name  Age     City
0  Alice   25       NY
1    Bob   30       LA
3  David   40  Chicago
```

In [3]:
```
# 2. Keeping the Last Occurrence of Duplicates
#By default drop_duplicates() retains the first occurrence of duplicates.
#If we want to keep the last occurrence we can use keep='last'.
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'David'],
    'Age': [25, 30, 25, 40],
    'City': ['NY', 'LA', 'NY', 'Chicago']
})

df_cleaned= df.drop_duplicates(keep='last')
print(df_cleaned)
```

```
    Name  Age     City
1    Bob   30       LA
2  Alice   25       NY
3  David   40  Chicago
```

In [4]:
```
#3. Dropping All Duplicates
#If we want to remove all rows that are duplicates, we can set keep=False.
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'David'],
    'Age': [25, 30, 25, 40],
    'City': ['NY', 'LA', 'NY', 'Chicago']
})
df_cleaned = df.drop_duplicates(keep=False)
print(df_cleaned)
```

```
    Name  Age     City
1    Bob   30       LA
3  David   40  Chicago
```

In [5]:
```
#4. Modifying the Original DataFrame Directly
#If we want to modify the DataFrame in place without
#creating a new DataFrame set inplace=True.
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'David'],
    'Age': [25, 30, 25, 40],
    'City': ['NY', 'LA', 'NY', 'Chicago']
})
df.drop_duplicates(inplace=True)
print(df)
```

```
    Name  Age     City
0  Alice   25       NY
1    Bob   30       LA
3  David   40  Chicago
```

In [6]:
```
#5. Dropping Duplicates Based on Partially Identical Columns
#Sometimes we might encounter situations where duplicates
#are not exact rows but have identical values in certain columns.
#For example after merging datasets we may
#want to drop rows that have the same values in a subset of columns.

import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Alice", "David", "Bob"],
    "Age": [25, 30, 25, 40, 30],
    "City": ["NY", "LA", "NY", "Chicago", "LA"]
}

df = pd.DataFrame(data)

df_cleaned = df.drop_duplicates(subset=["Name", "City"])

print(df_cleaned)
```

```
    Name  Age     City
0  Alice   25       NY
1    Bob   30       LA
3  David   40  Chicago
```

Pandas Change Datatype

In data analysis, ensuring that each column in a Pandas DataFrame has the correct data type is crucial for accurate computations and analyses. The most common way to change the data type of a column in a Pandas DataFrame is by using the astype() method. This method allows you to convert a specific column to a desired data type.

In [7]:
```
import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob', 'Eve', 'Charlie'],
        'Age': [25, 30, 22, 35, 28],
        'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
```

```
        'Salary': [50000, 55000, 40000, 70000, 48000]}

df = pd.DataFrame(data)

# Convert 'Age' column to float type
df['Age'] = df['Age'].astype(float)
print(df.dtypes)
```

```
Name       object
Age       float64
Gender     object
Salary      int64
dtype: object
```

In [8]:
```
#Converting a Column to a DateTime Type
#Sometimes, a column that contains date information may be stored as a string.
#You can convert it to the datetime type using the pd.to_datetime() function.
# Example: Create a 'Join Date' column as a string
df['Join Date'] = ['2021-01-01', '2020-05-22', '2022-03-15', '2021-07-30', '2020-11

# Convert 'Join Date' to datetime type
df['Join Date'] = pd.to_datetime(df['Join Date'])
print(df.dtypes)
```

```
Name                 object
Age                 float64
Gender               object
Salary                int64
Join Date    datetime64[ns]
dtype: object
```

In [9]:
```
#Changing Multiple Columns' Data Types
#If you need to change the data types of multiple columns at once,
#you can pass a dictionary to the astype() method,
#where keys are column names and values are the desired data types.
# Convert 'Age' to float and 'Salary' to string
df = df.astype({'Age': 'float64', 'Salary': 'str'})
print(df.dtypes)
```

```
Name                 object
Age                 float64
Gender               object
Salary               object
Join Date    datetime64[ns]
dtype: object
```

Drop Empty Columns in Pandas

Cleaning data is an essential step in data analysis. In this guide we will explore different ways to drop empty, null and zero-value columns in a Pandas DataFrame using Python. By the end you'll know how to efficiently clean your dataset using the dropna() and replace() methods.

Understanding dropna()

The dropna() function is a powerful method in Pandas that allows us to remove rows or columns containing missing values (NaN). Depending on the parameters used it can remove rows or columns where at least one value is missing or only those where all values are missing.

Syntax: DataFrameName.dropna(axis=0, how='any', inplace=False)

Parameters:

axis: axis takes int or string value for rows/columns. Input can be 0 or 1 for Integer and 'index' or 'columns' for String.

how: how takes string value of two kinds only ('any' or 'all'). 'any' drops the row/column if ANY value is Null and 'all' drops only if ALL values are null.

inplace: It is a boolean which makes the changes in the data frame itself if True.

In [10]:
```
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                   "Gender": ["", "", ""],
                   "Age": [0, 0, 0]})
df['Department'] = np.nan

print(df)
```

```
  FirstName Gender  Age  Department
0     Vipul           0         NaN
1    Ashish           0         NaN
2     Milan           0         NaN
```

In [11]:
```
# : Remove All Null Value Columns
#This method removes columns where all values are NaN.
#If a column is completely empty (contains only NaN values)
#it is unnecessary for analysis and can be removed using dropna(how='all', axis=1).
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                   "Gender": ["", "", ""],
                   "Age": [0, 0, 0]})

df['Department'] = np.nan

display(df)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

	FirstName	Gender	Age
0	Vipul		0
1	Ashish		0
2	Milan		0

In [12]:
```
# Replace Empty Strings with Null and Drop Null Columns
#If a column contains empty strings we need to replace them with NaN
#before dropping the column. Empty strings are not automatically recognized as miss
#in Pandas so converting them to NaN ensures they can be handled correctly. After c
#to remove columns that are entirely empty.
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                               "Gender": ["", "", ""],
                               "Age": [0, 0, 0]})

df['Department'] = np.nan
display(df)

nan_value = float("NaN")
df.replace("", nan_value, inplace=True)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

```
C:\Users\HP\AppData\Local\Temp\ipykernel_13880\2146771061.py:17: FutureWarning: Down
casting behavior in `replace` is deprecated and will be removed in a future version.
To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To o
pt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', Tru
e)`
  df.replace("", nan_value, inplace=True)
```

	FirstName	Age
0	Vipul	0
1	Ashish	0
2	Milan	0

In [13]:
```
#Replace Zeros with Null and Drop Null Columns
#If columns contain only zero values, we convert them to NaN before dropping them.
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                           "Gender": ["", "", ""],
                           "Age": [0, 0, 0]})

df['Department'] = np.nan
display(df)

nan_value = float("NaN")
df.replace(0, nan_value, inplace=True)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

	FirstName	Gender
0	Vipul	
1	Ashish	
2	Milan	

In [14]:
```
#Replace Both Zeros and Empty Strings with Null and Drop Null Columns
#To clean a dataset fully we may need to replace both zeros and empty strings.
import numpy as np
import pandas as pd

df = pd.DataFrame({'FirstName': ['Vipul', 'Ashish', 'Milan'],
                           "Gender": ["", "", ""],
                           "Age": [0, 0, 0]})

df['Department'] = np.nan
display(df)
```

```
nan_value = float("NaN")
df.replace(0, nan_value, inplace=True)
df.replace("", nan_value, inplace=True)

df.dropna(how='all', axis=1, inplace=True)

display(df)
```

	FirstName	Gender	Age	Department
0	Vipul		0	NaN
1	Ashish		0	NaN
2	Milan		0	NaN

```
C:\Users\HP\AppData\Local\Temp\ipykernel_13880\3018853403.py:15: FutureWarning: Down
casting behavior in `replace` is deprecated and will be removed in a future version.
To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To o
pt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', Tru
e)`
  df.replace("", nan_value, inplace=True)
```

	FirstName
0	Vipul
1	Ashish
2	Milan

In [15]:
```
# String Manipulations in Pandas DataFrame
import pandas as pd
import numpy as np

data = { 'Name': ['Ratna', 'Sofia', 'Hiroshi', 'Marta', 'Smruti', np.nan, 'Elena'],
         'City': ['Balasore', 'Delhi', 'Hyderabad', 'Mumbai', 'Bhubaneswar', 'Keral

df = pd.DataFrame(data)
print(df)
```

```
     Name         City
0    Ratna     Balasore
1    Sofia        Delhi
2  Hiroshi    Hyderabad
3   Marta       Mumbai
4  Smruti  Bhubaneswar
5     NaN       Kerala
6   Elena          Goa
```

In [16]:
```
print(df.astype('string'))
```

```
      Name         City
0    Ratna     Balasore
1    Sofia        Delhi
2  Hiroshi    Hyderabad
3    Marta       Mumbai
4   Smruti  Bhubaneswar
5     <NA>       Kerala
6    Elena          Goa
```

String Operations in Pandas

Below are the commonly used string manipulation methods in Pandas, explained with short examples.

1. lower(): This method converts every character in the column to lowercase, ensuring consistent text formatting.

In [17]:
```
print(df['Name'].str.lower())
```

```
0      ratna
1      sofia
2    hiroshi
3      marta
4     smruti
5        NaN
6      elena
Name: Name, dtype: object
```

In [18]:
```
print(df['Name'].str.upper())
```

```
0      RATNA
1      SOFIA
2    HIROSHI
3      MARTA
4     SMRUTI
5        NaN
6      ELENA
Name: Name, dtype: object
```

In [19]:
```
# 3. strip(): This method removes unwanted leading and
#trailing spaces from each string to clean the data.
print(df['Name'].str.strip())
```

```
0      Ratna
1      Sofia
2    Hiroshi
3      Marta
4     Smruti
5        NaN
6      Elena
Name: Name, dtype: object
```

In [20]:
```
# 4. split(): This method splits each string into a list of parts based on a given
print(df['Name'].str.split('a'))
```

```
0    [R, tn, ]
1     [Sofi, ]
2    [Hiroshi]
3    [M, rt, ]
4     [Smruti]
5          NaN
6     [Elen, ]
Name: Name, dtype: object
```

In [21]:
```
# 5. len(): This method calculates and returns the character length of each string
#in the column.
print(df['Name'].str.len())
```

```
0    5.0
1    5.0
2    7.0
3    5.0
4    6.0
5    NaN
6    5.0
Name: Name, dtype: float64
```

In [24]:
```
#6. cat(): This method concatenates all strings in the column into
#a single string using a chosen separator.
print(df['Name'].str.cat(sep=','))
```

```
Ratna,Sofia,Hiroshi,Marta,Smruti,Elena
```

In [25]:
```
#7. get_dummies(): This method converts each unique string into a
#separate one-hot encoded column for modeling.
print(df['City'].str.get_dummies())
```

```
   Balasore  Bhubaneswar  Delhi  Goa  Hyderabad  Kerala  Mumbai
0         1            0      0    0          0       0       0
1         0            0      1    0          0       0       0
2         0            0      0    0          1       0       0
3         0            0      0    0          0       0       1
4         0            1      0    0          0       0       0
5         0            0      0    0          0       1       0
6         0            0      0    1          0       0       0
```

In [26]:
```
df
```

Out[26]:

	Name	City
0	Ratna	Balasore
1	Sofia	Delhi
2	Hiroshi	Hyderabad
3	Marta	Mumbai
4	Smruti	Bhubaneswar
5	NaN	Kerala
6	Elena	Goa

In [28]:
```
# 8. startswith(): This method checks whether each string begins with the specified
print(df['Name'].str.startswith('E'))
```

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6     True
Name: Name, dtype: object
```

In [30]:
```
#9.endswith(): This method checks whether each string ends with the specified suffi
print(df['Name'].str.endswith('a'))
```

```
0     True
1     True
2    False
3     True
4    False
5      NaN
6     True
Name: Name, dtype: object
```

In [31]:
```
#replace(): This method replaces occurrences of a specific
#substring or pattern with a new value.
print(df['Name'].str.replace('Elena', 'Emily'))
```

```
0      Ratna
1      Sofia
2    Hiroshi
3      Marta
4     Smruti
5        NaN
6      Emily
Name: Name, dtype: object
```

In [32]:
```
#. repeat(): This method duplicates each string a given number of times.
print(df['Name'].str.repeat(2))
```

```
0        RatnaRatna
1        SofiaSofia
2    HiroshiHiroshi
3        MartaMarta
4      SmrutiSmruti
5               NaN
6        ElenaElena
Name: Name, dtype: object
```

In [33]:
```
#. count(): This method counts how many times a
#specific substring or pattern appears in each string.
print(df['Name'].str.count('a'))
```

```
0    2.0
1    1.0
2    0.0
3    2.0
4    0.0
5    NaN
6    1.0
Name: Name, dtype: float64
```

In [34]:
```
#  find(): This method returns the index of the first occurrence of a
#pattern within each string.
print(df['Name'].str.find('a'))
```

```
0    1.0
1    4.0
2   -1.0
3    1.0
4   -1.0
5    NaN
6    4.0
Name: Name, dtype: float64
```

In [35]:
```
#  findall(): This method returns a list of all occurrences of a pattern found in e
print(df['Name'].str.findall('a'))
```

```
0    [a, a]
1       [a]
2        []
3    [a, a]
4        []
5       NaN
6       [a]
Name: Name, dtype: object
```

In [36]:
```
#islower(): This method checks whether all characters in each string are lowercase.
print(df['Name'].str.islower())
```

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Name, dtype: object
```

In [37]:
```
#  isupper(): This method checks whether all characters in each string are uppercas
print(df['Name'].str.isupper())
```

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Name, dtype: object
```

In [38]:
```
#  isnumeric(): This method checks whether each string contains only numeric charac
print(df['Name'].str.isnumeric())
```

```
0    False
1    False
2    False
3    False
4    False
5      NaN
6    False
Name: Name, dtype: object
```

In [39]:
```
#  swapcase(): This method swaps uppercase letters to lowercase and
#lowercase letters to uppercase for each string.
print(df['Name'].str.swapcase())
```

```
0      rATNA
1      sOFIA
2    hIROSHI
3      mARTA
4     sMRUTI
5        NaN
6      eLENA
Name: Name, dtype: object
```

In [40]:
```
import pandas as pd
sports = pd.Series(['Virat', 'azam', 'fiNch', 'ShakiB', 'STOKES', 'KAne'])
print(sports)
```

```
0     Virat
1      azam
2     fiNch
3    ShakiB
4    STOKES
5      KAne
dtype: object
```

In [41]:
```
print("Upper Case:")
print(sports.str.upper())
```

```
Upper Case:
0     VIRAT
1      AZAM
2     FINCH
3    SHAKIB
4    STOKES
5      KANE
dtype: object
```

In [43]:
```
print("Lower Case:")
print(sports.str.lower())
```

```
Lower Case:
0     virat
1      azam
2     finch
3    shakib
4    stokes
5      kane
dtype: object
```

Pandas: Detect Mixed Data Types and Fix it

What are mixed types in Pandas columns?

As you know, Pandas data frame can have multiple columns, thus when a certain column doesn't have a specified kind of data, i.e., doesn't have a certain data type, but contains mixed data, i.e., numeric as well as string values, then that column is tend to have mixed data type.

Causes of mixed data types

Missing Values (NaN)

Inconsistent Formatting

Data Entry Errors

Missing Values (NaN):

A floating-point value that represents undefined or unrepresentable data is known as NaN. The most common use case of NaN occurrence is the 0/0 case, which leads to mixed data types and ultimately leads to incorrect results.

Inconsistent Formatting:

The inconsistent formatting in the Pandas data frame is observed due to the cells with wrong format. Thus, it is crucial to transform each cell of column to a correct format.

Data Entry Errors:

There occurs various instances when the user makes a mistake while entering the data in a column in Pandas data frame. It can be any error, entering string data in numeric type column or leaving null value in the column or anything. Such errors can also lead to mixed data types and thus need to be fixed.

How to identify mixed types in Pandas columns

You might have used info() function to detect the data type of Pandas data frame, but using info() function is not possible in case of mixed data types. For detecting the mixed data types, you need to traverse each column of Pandas data frame, and get the data type using api.types.infer_dtypes() function.

Syntax:

for column in data_frame.columns:

print(pd.api.types.infer_dtype(data_frame[column]))

Here,

data_frame: It is the Pandas data frame for which you want to detect if it has mixed data types or not.

In [44]:
```
# Python program to detect mixed data types in Pandas data frame

# Import the library Pandas
import pandas as pd

# Create the pandas DataFrame
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=[

# Traverse data frame to detect mixed data types
for column in data_frame.columns:
    print(column,':',pd.api.types.infer_dtype(data_frame[column]))
```

```
Name : string
Age : mixed-integer
```

How to deal with mixed types in Pandas columns

For fixing the mixed data types in Pandas data frame, you need to convert entire column into one data type. This can be done using astype() function or to_numeric() function.

Using astype() function:

A crucial function in Pandas which is used to cast an object to a specified data type is known as astype() function. In this way, we will see how we can fix mixed data types using astype() function.

Syntax:

data_frame[column] = data_frame[column].astype(int)

Here,

data_frame: It is the Pandas data frame for which you want to fix mixed data types.

column: It defines all the columns of the Pandas data frame.

int: Here, int is the data type in which you want to transform type of each column of Pandas data frame. You can also use str, float, etc. here depending on which data type you want to transform.

In [45]:
```
# Python program to fix mixed data types using astype() in Pandas data frame

# Import the library Pandas
import pandas as pd

# Create the pandas DataFrame
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=[

# Transforming mixed data types to single data type
data_frame[column] = data_frame[column].astype(int)

# Traverse data frame to detect data types after fix
for column in data_frame.columns:
    print(column,':',pd.api.types.infer_dtype(data_frame[column]))
```

```
Name : string
Age : integer
```

In [46]:
```
# Python program to fix mixed data types using to_numeric() in Pandas data frame

# Import the library Pandas
import pandas as pd

# Create the pandas DataFrame
data_frame = pd.DataFrame( [['tom', 10], ['nick', '15'], ['juli', 14.8]], columns=[

# Transforming mixed data types to single data type
data_frame[column] = data_frame[column].apply(lambda x: pd.to_numeric(x, errors = '

# Traverse data frame to detect data types after fix
for column in data_frame.columns:
  print(pd.api.types.infer_dtype(data_frame[column]))
```

```
string
floating
```

```
C:\Users\HP\AppData\Local\Temp\ipykernel_13880\810235737.py:10: FutureWarning: errors='ignore' is deprecated and will raise in a future version. Use to_numeric without passing `errors` and catch exceptions explicitly instead
  data_frame[column] = data_frame[column].apply(lambda x: pd.to_numeric(x, errors = 'ignore'))
```

In [47]:
```
# Pandas DataFrame corr() Method:
import pandas as pd

df = {
    "Array_1": [30, 70, 100],
    "Array_2": [65.1, 49.50, 30.7]
}

data = pd.DataFrame(df)

print(data.corr())
```

```
          Array_1   Array_2
Array_1  1.000000 -0.990773
Array_2 -0.990773  1.000000
```

In [ ]:
```

```