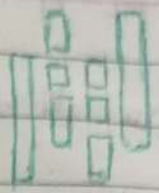
# Pandas



Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language

Basic data structure.
Pandas provides two types of classes for handling data:

1 series : a one-dimensional labeled array holding data of any type such as Integers, strings, Python objects etc

2 Dataframe :- a two-dimensional data structure that holds data like a 2-D array. or a table with rows and columns.

Creating ~~dataframes~~ series.

1 from a list

```
import pandas as pd
data = [10, 20, 30, 40]
series = pd.series(data)
print(series)
```

data = [10, 20, 30, 40]

series =
| | |
|---|---|
| 0 | 10 |
| 1 | 20 |
| 2 | 30 |
| 3 | 40 |

dtype : int64

2. From a dictionary.

```
data = {'a': 1, 'b': 2, 'c': 3}
series = pd.Series(data)
print(series)
```

data = {'a':1, 'b':2, 'c':

series =
a 1
b 2
c 3
dtype : int64

3. With custom index

```
data = [100, 200, 300]
indexes = ['x', 'y', 'z']
series = pd.Series(data, index = indexes)
print(series)
```

data = [10, 20, 30, 40]
indexes = ['w', 'x', 'y', 'z']

series =
w 10
x 20
y 30
z 40
dtype : int64

Creating dataframes

1 From Dictionary

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 45],
    'City': ['New York', 'Los Angles', 'Chicago']
}
```

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

dataframe :-

| | Name | Age | City |
|---|---|---|---|
| 0 | Alice | 25 | New York |
| 1 | Bob | 30 | Los Angeles |
| 2 | Charlie | 45 | Chicago |

2. From list of lists

```
data = [
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'Los Angeles'],
    ['Charlie', 45, 'Chicago']
    ]
df = pd.DataFrame(data, columns = ['Name', 'Age', 'City'])
print(df)
```

dataframe: same as above table

3. From a CSV file or Excel file

```
df = pd.read_csv('data.csv')
print(df)
```

```
df = pd.read_excel('data.xlsx', sheet_name = 'Sheet1')
print(df)
```

## head() & tail()

In Pandas, the head() and tail() methods to ~~pandas~~ are used to retrieve the first and last rows of a DataFrame or series, respectively. By default, both methods return 5 rows, but you can specify the no. of rows to display.

Ex: import pandas as pd

```
data = {
    "Name": ["Alice", "Bob", "Charlie", "David", "Eve",
             "Frank", "Grace"],
    "Age": [24, 27, 22, 32, 29, 25, 28],
    "City": ["NY", "LA", "SF", "NY", "LA", "SF", "NY"]
}
df = pd.DataFrame(data)

# Display the first 3 rows
print(df.head(3))

# Display the last 2 rows
print(df.tail(2))
```

→

| | Name | Age | City |
|---|---|---|---|
| 0 | Alice | 24 | NY |
| 1 | Bob | 27 | LA |
| 2 | Charlie | 22 | SF |

↳

| | Name | Age | City |
|---|---|---|---|
| 5 | Frank | 25 | SF |
| 6 | Grace | 28 | NY |

# Info() & describe()

In Pandas, the info() and describe() are two essential methods used to understand the structure and summary statistics of a DataFrame

1. Info() : Includes
   - no. of rows and columns.
   - Column names and their data types
   - non-null value counts for each column
   - Memory usage of the DataFrame.

Ex:-
```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'Salary': [50000, 60000, 70000]}

df = pd.DataFrame(data)

df.info()
```

output:
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    3 non-null      object
 1   Age     3 non-null      int64
 2   Salary  3 non-null      int64
dtypes: int64(2), object(1)
memory usage: 200.0+ bytes
```

2 describe () : This method generates summary statistics for numerical columns by default, such as:

- Count, mean, standard deviation, minimum, maximum
- Quartiles (25%, 50%, 75%)

Ex : Use df.describe() on previous example

Output :-

| | age | salary |
|---|---|---|
| count | 3.000000 | 3.0 |
| mean | 37.666667 | 55000.0 |
| std | 14.189138 | 5000.0 |
| min | 25.000000 | 50000.0 |
| 25% | 30.000000 | 52500.0 |
| 50% | 35.000000 | 55000.0 |
| 75% | 44.000000 | 57500.0 |
| max | 53.000000 | 60000.0 |

df.describe (include = 'all')
for all columns. (non-numerical also)
such as:-

- unique
- count
- top value
- frequency.

Accessing columns and rows.

Accessing columns.

1 using column name:

```
df['column_name']      for series
df[['column1','column2']]   for Dataframe
```

2. using dot notation

```
df.column_name
```

Accessing rows

1 using index location (iloc) :- Access by integer position.

```
df.iloc[0]
df.iloc[0:3]
```

2 using label (loc) :- Access by index label

```
df.loc['row-label']
df.loc['label1':'label3']
```

3 using conditional filtering:

```
df[df['column_name']>10]
```

Accessing specific Rows and columns together

1 using iloc :-

df.iloc[0,2] value at first row, second column
df.iloc[0:2,1:] subset of rows and columns

2. using loc :-

df.loc['row_label', 'column_name'] specific value
df.loc['label1':'label2',['column1', 'column2']] subset of rows and c

Data types and type conversion.

dtypes :- displays the data types of each column in a Dataframe or the data type of a series.

Ex :- import pandas as pd.

```
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30],
        'salary': [50000.5, 60000.0]}

df = pd.Dataframe(data)
print(df.dtypes)
```

Output :-

| | |
|---|---|
| Name | object |
| Age | int64 |
| salary | float64 |

dtype : object.

astype() :- Converts the data type of a Series or Dataframe column to a specific type.

Ex :- # Converts 'Age' to float

```
df['Age'] = df['Age'].astype(float)
print(df.dtypes)
```

output :-

```
No Name      object
Age          float64
Salary       float64
dtype: object
```

Multiple Column Conversion.

Ex :-

```
df = df.astype({'Age': 'int', 'Salary': 'int'})
print(df.dtypes)
```

output :-

```
Name      object
Age       int32
Salary    int32
dtype: object
```

In: Intermediate level.

Indexing and slicing.

Indexing :- used to access specific rows, columns, or elements in a DF or series.

Single-column selection :-

```
df = pd.DataFrame({'A':[1,2,3], 'B':[4,5,6]})
print(df['A'])
```

row-selection

using .loc (label-based indexing):

```
print(df.loc[0])
```

using .iloc (integer-based indexing):

```
print(df.iloc[0])
```

Slicing :- allows to extract subsets of rows or colu

Row-slicing

using .iloc :

```
print(df.iloc[0:2])
```

- using .loc :

```
print(df.loc[0:1])
```

Column slicing

```
print(df.iloc[:, 0:2]) # All rows, column 0 to 1
```

Mixed slicing

```
print(df.loc[0:1, 'A':'B'])
```

Boolean Indexing :- filter rows based on condn

```
print(df[df['A'] > 1])
```

Fancy Indexing :- selects specific colms or rows using list

```
print(df.iloc[[0, 2], [0, 1]])
```

## Handling Missing data:-

Handling missing values in pandas is a common task in data processing. Here are three approaches to deal with missing values effectively:

### 1. Removing missing values

- You can drop rows or columns with missing values using dropna().

Ex:
```
df = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})
df_dropped_rows = df.dropna()
df_dropped_clm = df.dropna(axis = 1)
```

### 2 Filling Missing Values

- Use fillna() to replace missing values with a specific value, mean, median or other strategies.

Ex:-
```
df_filled_constant = df.fillna(0)

df_filled_mean = df.fillna(df.mean())

df_filled_ffill = df.fillna(method = 'ffill')
```

3 Identifying missing values

You can detect missing values using:

```
import pandas as pd

df.isnull()
df.isnull().sum()
```

Filtering & conditional selection.

1 Basic Filtering

You can filter rows based on a condition applied to a column.

Ex:-
```
import pandas as pd

data = {'Name': ['Name', 'Alice', 'Bob', 'Charlie', 'David'],
        'Age': [24, 27, 22, 32],
        'City': ['Indore', 'Mumbai', 'Delhi', 'Indore']}

df = pd.DataFrame(data)

filtered_df = df[df['Age'] > 25]
print(filtered_df)
```

2 Multiple Conditions

Use logical operators (&, |, ~) for combining conditions :-

- & for AND
- | for OR
- ~ for NOT

Ex:- filtered_df = df[(df['Age'] > 25) & (df['City'] == 'Indor
'')]
print (filtered_df)

filtered_df = df[(df['Age'] < 25) | (df['City'] ==
'Mumbai')]
print (filtered_df)

filtered_df = df[~(df['City'] == 'Indor')]
print (filtered_df)

3.

3. Using .query()

The .query() method provides a more readable way to filter data.

Ex:- filtered_df = df.query('Age > 25')
print(filtered_df)

filtered_df = df.query('Age > 25 and City == "Indo
print(filtered_df)

4. Filtering with .isin()

Use .isin() to filter rows based on multiple values in a column.

Ex:- filtered_df = df[df['City'].isin(['Indore', 'Mumbai'])]
print(filtered_df)

5. Filtering with .str method.

for string-based filtering, use .str accessor.

Ex:- filtered_df = df[df['City'].str.startswith('I')]
print(filtered_df)

filtered_df = df[df['Name'].str.contains('i')]
print(filtered_df)

6. Conditional Selection with .loc

Use .loc for more explicit filtering and selection

Ex: filtered_df = df.loc[df['Age'] > 25, ['Name', 'City']]
print(filtered_df)

Sorting :

1. Sorting by values (sort_values)

This method sorts a DataFrame or Series by one or more columns or values

Ex: ~~Impo~~ data = {'Name': ['Alice', 'Bob', 'Charlie'],
'Age': [25, 30, 20],
'Score': [85, 90, 88]}

df = pd.DataFrame(data)

Sorted_by_Age = df.sort_values(by='Age')

Sorted_by_score_and_age = df.sort_values(by=['Score', 'Age'], ascending=[False, True])

2. Sorting by Index (sort_index)

This method sorts the DataFrame or Series by the index.

Ex:- sorted_by_index = df.sort_index()

Sorted_by_index_desc = df.sort_index(ascending=False)

3. In-place sorting.

You can sort the DataFrame in place by setting inplace=True

Ex:- df.sort_values(by='Age', inplace=True)

# Aggregation and Grouping.

for analysing and summarising data.

## 1. Grouping data.

The .groupby() method is used to group data based on one or more columns. It creates a "grouped" object that can be further processed.

Ex:
```
import pandas as pd

data = {'category': ['A', 'B', 'A', 'B', 'A'],
        'values': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)

grouped = df.groupby('category')
```

## 2. Aggregation.

Aggregation involves applying functions (like sum, mean, etc) to grouped data. You can use built-in fun^n or custom ones.

Ex:
```
aggregated = grouped['values'].sum()
print(aggregated)
```

Output:-

| Category | |
|---|---|
| A | 90 |
| B | 60 |

3. Multiple Aggregations

You can apply multiple aggregation function using .agg()

```
Ex: aggregated = grouped['values'].agg(['sum', 'mean', 'max'])
    print(aggregated)
```

Output:

| Category | sum | mean | max |
|---|---|---|---|
| A | 90 | 30.0 | 50 |
| B | 60 | 30.0 | 40 |

4. using groupby with apply.

For more complex operations, use .apply() to apply a custom function to each group.

```
Ex:- def calculate_range(group):
        return group['values'].max() - group['values'].min()

     result = df.groupby('category').apply(calculate_range)
     print(result)
```

Output:-

Category

| | |
|---|---|
| A | 40 |
| B | 20 |

Name: values, dtype: float64

Combining data:

Combining data in pandas can be done using various methods depending on the structure and requirements of your data.

1 Concatenation

You can stack Dataframes either vertically (row-wise) or horizontally (column-wise) using pd.concat().

Ex:
```
import pandas as pd

df1 = pd.DataFrame({'A': [1,2], 'B': [3,4]})
df2 = pd.Dataframe({'A': [5,6], 'B': [7,8]})
# Vertically concat^
result = pd.concat([df1, df2], axis = 0)
# Horizontally concat^
result_horizontal = pd.concat([df1, df2], axis = 1)
```

2 Merging.

Use pd.merge() to combine Dataframes based on a ~~column~~ common column or index, similar to SQL Joins.

Ex:
```
df1 = pd.Dataframe({'Key': ['A', 'B'], 'value1': [1,2]})
df2 = pd.Dataframe({'Key': ['A', 'B'], 'value2': [3,4]})
```

result = pd.merge(df1, df2, on='key', how='inner')

3. Joining :-

For combination combining Dataframes based on their index, we df.join().

Ex :- df1 = pd.DataFrame({'value1': [1,2]}, 'index' = ['A', 'B'])
df2 = pd.DataFrame({'value2': [3, 4]}, index = ['A', 'B'])

result = df1.join(df2)

Output :- Concatination :-

V=

| | A | B |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 2 | 4 |
| 0 | 5 | 7 |
| 1 | 6 | 8. |

H=

| | A | B | A |
|---|---|---|---|
| 0 | 1 | 3 | 5 |
| 1 | 2 | 4 | 6 |

Merging :-

| | key | value1 | value2 |
|---|---|---|---|
| 0 | A | 1 | 3 |
| 1 | B | 2 | 4 |

Joining :-

| | value1 | value2 |
|---|---|---|
| A | 1 | 3 |
| B | 2 | 4 |

String operation on columns

1 convert to lowercase or uppercase

```
df = pd.DataFrame ({'Name': ['Alice', 'Bob', 'Charlie']})
df['Name_lower'] = df['Name'].str.lower()
df['Name_upper'] = df['Name'].str.upper()
```

2 strip whitespace

```
df['Name_stripped'] = df['Name'].str.strip()
```

3. Substring Extraction

```
df['first_3_chars'] = df['Name'].str[:3]
df['last_2_chars'] = df['Name'].str[-2:]
```

4. Find and replace

```
df['Name_replaced'] = df['Name'].str.replace('a', 'X',
                      case=False)
```

5. check for substring

```
df['contains_a'] = df['Name'].str.contains('a', case=False)
```

6. Split strings

```
df['split_name'] = df['Name'].str.split(' ')
```

7. Concatenate strings

```
df['Greeting'] = 'Hello, ' + df['Name']
```

8. Length of strings.

```
df['Name_length'] = df['Name'].str.len()
```

9. Check for Patterns (Regex)

```
df['starts_with_A'] = df['Name'].str.startswith('A')
df['Ends_with_e'] = df['Name'].str.endswith('e')
```

10. Remove Non-Alphanumeric Characters.

```
df['cleaned_Name'] = df['Name'].str.replace(r'[^a-zA-Z0-
', regex = True)
```

## Applying Custom Functions

To apply a custom function in Pandas, you can use methods like apply(), applymap(), or map() depending on whether you're working with a DataFrame, Series, or individual elements.

1. Using apply().

The apply() method is versatile and works on rows or columns of a DataFrame or on a Series.

```
data = {'A': [1,2,3], 'B': [4,5,6]}
df = pd.DataFrame(data)

def custom_function(x):
    return x*2

df['A'] = df['A'].apply(custom_function)

df['Sum'] = df.apply(lambda row: row['A'] +
                     row['B'], axis=1)
```

2. Using applymap().

The applymap() method applies a function element-wise to the entire DataFrame.

```
def square(x):
    return x**2

df = df.applymap(square)
```

3. using map() for series.

The map() method is specifically for series and is ideal for element-wise transformations.

```
s = pd.Series([1, 2, 3, 4])

squared = s.map(lambda x: x**2)
```

## Advanced level

- Pivot tables.

Pivot tables in pandas are a powerful tool for summarizing and analyzing data. They allow you to reshape and aggregate data in a tabular format, similar to Excel's pivot tables.

Key parameters of pandas.pivot_table().

data : The DataFrame to pivot.
Index : Rows of the pivot table.
Columns : Columns of the pivot table
values : values to aggregate
aggfunc : Aggregation function. (default is mean)

Ex: Creating a pivot table.

```
import pandas as pd

data = {
    'City': ['Indore', 'Bhopal', 'Indore', 'Bhopal', 'Indore'],
    'Product': ['A', 'A', 'B', 'B', 'A'],
    'Sales': [200, 150, 300, 400, 250]
}
df = pd.DataFrame(data)
```

```
pivot = pd.pivot_table(
    df,
    values='Sales',
    index='City',
    columns='Product',
    aggfunc='sum',
    fill_value=0   # replace NaN with 0
)
print(pivot)
```

Output:-

| product | A | B |
|---|---|---|
| City | | |
| Bhopal | 150 | 400 |
| Indore | 450 | 300 |

Multiindex Dataframes.

A Multiindex in pandas is a powerful feature that allows you to work with hierarchical or multi-level indexing in Dataframes or series. It enables you to represent and manipulate data with multiple dimensions or levels of labels, making it easier to handle complex datasets.

What is a multiindex?

- A MultiIndex is an advanced index structure in pandas that can have multiple levels (or tiers) of indexing.
- It is useful for organizing data in a hierarchical way, such as grouping by multiple keys or representing multi-dimensional data in a tabular format.

Creating a multiIndex Dataframe

1. from a list of tuples.

```
import pandas as pd

index = pd.MultiIndex.from_tuples([('A', 1),
                ('A', 2), ('B', 1), ('B', 2)],
names = ['Group', 'Subgroup'])

df = pd.Dataframe({'Value': [10, 20, 30, 40]},
                  index = index)
print(df)
```

output:-

| | | Group | Subgroup | Value |
|---|---|---|---|---|
| | | A | 1 | 10 |
| | | | 2 | 20 |
| | | B | 1 | 30 |
| | | | 2 | 40 |

using set_index.

```
data = {'Group': ['A', 'A', 'B', 'B'], 'Subgroup': [1,2,
        'Value': [10, 20, 30, 40]}

df = pd.DataFrame(data).set_index(['Group', 'Subgro
print(df).
```

Accessing Data in Multiindex.

- Accessing rows by levels:

```
print(df.loc['A'])
print(df.loc[('A', 1)])
```

- Reset index to flatten:

```
df_reset = df.reset_index()
print(df_reset)
```

# Windows functions:

~~Rolling~~ Pandas window functions provide a flexible framework for performing rolling and expanding calculations on data. These functions allow for efficient computation of statistics over sliding or expanding windows of data, making them essential for time series analysis and other data manipulation tasks.

## Rolling window functions.

Rolling window functions apply a calculation over a fixed-size sliding window of data, for example:- To calculate the rolling sum of series with a window size of 2:

Ex:
```
import pandas as pd

s = pd.Series(range(5))
rolling_sum = s.rolling(window = 2).sum()
print(rolling_sum)
```

Output :-

```
0   NaN
1   1.0
2   3.0
3   5.0
4   7.0
dtype: float64
```

## Expanding Window Functions

Expanding window functions calculate statistics over an expanding window, which includes all data points up to the current point. For example, to calculate the expanding sum:

```
Ex: expanding_sum = s.expanding().sum()
    print(expanding_sum)
```

Output:-

```
0    0.0
1    1.0
2    3.0
3    6.0
4   10.0
dtype: float64
```

## Exponentially Weighted Window Functions.

Exponentially weighted window function apply weights that decreases exponentially. This is useful for giving more importance to recent data points. For example, To calculate an exponentially weighted mean:

```
Ex:
ewm_mean = s.ewm(span=2).mean()
print(ewm_mean)
```

output :-

```
0  0.000000
1  0.666667
2  1.555556
3  2.518519
4  3.506173
dtype: float64
```

## Custom Window Function.

Pandas also allows for custom window functions using the apply() method. For example, to calculate the mean absolute deviation:

```
Ex.  def mad(x):
         return np.fabs(x - x.mean()).mean()

     rolling_mad = s.rolling(window=4).apply(mad, ra
                                    raw=True)
     print(rolling_mad)
```

Output :-

```
0    NaN
1    NaN
2    NaN
3    1.0
4    1.0
dtype: float64
```

# Working with time-series Data

When working with time series data in Python, it's common to encounter dates and times represented as strings or numbers. Converting these into proper datetime objects allows for more efficient and accurate time-based operations. The pandas library provides powerful tools to handle such conversions.

## Using Pandas.to_datetime()

versatile tool for converting date strings or numbers into datetime objects.

Ex 1: Converting Date strings to Time Series

```
import pandas as pd

dt_series = pd.Series ['28 July 2020', '16 January 2017',
            '29 February 2016 18:14']

time_series = pd.to_datetime (dt_series)

print ("series of date strings : \n", dt_series)
print ("series of date strings after being converted
  to a time series : \n", time_series).
```

Ex 2: Converting Numeric Date formats

```
import pandas as pd

dt_series = pd.Series(['20200728', '20130116', '20160229 181431'])

time_series = pd.to_datetime(dt_series)

print("series of numeric date formats :")
print(dt_series)
print("\nSeries of numeric date formats after being converted to a time series :")
print(time_series)
```

Ex 3: Mixed Date formats

```
dt_series = pd.Series(['28 July 2020', '2013-01-16', '20160229 18:14', '5/03/2019 2215', '2015/12/04 09:23'])

time_series = pd.to_datetime(dt_series)

print("series of mixed date formats :")
print(dt_series)
print("\nSeries of mixed date formats after being converted to a time series :")
print(time_series)
```

Practical Application.

~~Resha~~. Resampling.

Ex:

```
data = {'value': [10, 20, 30, 40, 50]}
index = pd.date_range(start='2020-01-01', periods=5,
                      freq='D')
df = pd.DataFrame(data, index=index)
monthly_data = df.resample('M').sum()

print("Original DataFrame")
print(df)
print("\nResampled DataFrame (Monthly):")
print(monthly_data)
```

# Memory Optimization Techniques

When working with large datasets, memory usage becomes a concern. Pandas is powerful, but it can be memory inefficient if you're not careful. These optimization techniques help reduce RAM usage, improve speed, and avoid crashes.

## Why optimize?

- large CSV/Excel files can use hundreds of MBs or even GBs of memory when loaded.
- Default data types (like object) are inefficient.
- Every optimization can lead to faster processing, especially on limited hardware.

## Techniques to optimize Memory usage

1. Use dtype while reading data.
   By default Pandas tries to guess data type - often incorrectly.

Ex:-

```
df = pd.read_csv('sales.csv', dtype={'CustomerID': 'Int32',
    'Region': 'category'})
```

- Int32 uses less memory than Int64
- Category uses much less memory than object (strings).

2. Convert object columns to category.
strings take up a lot of memory. If a column has repeating strings (e.g. "male", "female"), convert it.

Ex:
```
df['Gender'] = df['Gender'].astype('category')
```

- Reduces memory by up to 90% for that column

3. Downcast numeric types.
Default numeric types are often int64 or float64.
Downcast them:

Ex:
```
df['Age'] = pd.to_numeric(df['Age'], downcast='integer')
df['Salary'] = pd.to_numeric(df['Salary'], downcast='float')
```

- int8, int16, float32 are more memory efficient.

4. Use memory_usage() to inspect
You can check how much memory each column uses:

```
df.memory_usage(deep=True)
```

Or the total usage:
```
df.info(memory_usage='deep')
```

5. Read only necessary columns
If you don't need all columns:

Ex: df = pd.read_csv('bigdata.csv', usecols = ['CustomerID', 'Amount'])

6. Process data in chunks (for huge files).
Don't load everything into memory:

Ex:- chunks = pd.read_csv('bigdata.csv', chunksize = 10000)
for chunk in chunks:
process(chunk)

7. Use inplace = True when possible
Avoid creating copies of DataFrames unnecessarily:

Ex:- df.drop(columns = ['Unused'], inplace = True)

8. Drop unnecessary columns early.
Columns like IDs or descriptions may not be useful:

Ex: df.drop(columns = ['Description', 'Unnamed: 0'], inplace = True)

Example Memory Optimization flow:

Ex: df = pd.read_csv('data.csv')

```
for col in df.select_dtypes('object').columns:
    df[col] = df[col].astype('category')

for col in df.select_dtypes(include=['float']).columns:
    df[col] = pd.to_numeric(df[col], downcast='float')

for col in df.select_dtypes(include=['int']).columns:
    df[col] = pd.to_numeric(df[col], downcast='integer')
```

# Performance (vectorized operations vs loops).

## vectorized operations vs python loops (in Numpy).

### What does it mean?

- Vectorized operations use Numpy's optimized backend to apply operations to entire arrays at once, rather than element-by-element.
- python loops use slow, interpreted logic that applies one operation at a time.

### Why are vectorized operations Better?

| Feature | Vectorized (Numpy) | Python loops |
|---|---|---|
| Speed | Extremely fast (C-based) | Much slower (Python only) |
| Syntax | short and clean | verbose and error-prone |
| Memory use | Efficient | often inefficient |
| parallelization | Supported internally | Not used. |

Ex1 : Element-wise Array Addition

* vectorized. (numpy)

```
import numpy as np
a = np.arange(1_000_000)
b = np.arange(1_000_000)
c = a+b.
```

Python loops

```
c = []
for i in range (1,000,000):
    c.append (a[i] + b[i])
```

Time comparision.

```
import time

start = time.time()
c = a + b
print("Numpy", time.time() - start)

start = time.time()
c = []
for i in range (len(a)):
    c.append (a[i] + b[i])
print("loop:", time.time() - start)
```

Output (approx):-

Numpy :- 0.002 sec.

loop :- 0.8 - ~~0.9~~ 1.5 sec.

Ex 2: Dot Product of matrices

Vectorized (Fast)

```
result = np.dot(A, B)
```

loops (slow)

```
result = [[sum(A[i][k] * B[k][j] for k in range(len(B)))
           for j in range(len(B[0]))]
           for i in range(len(A))]
```

Real use case: Why it matters in ML.

On machine learning:-

- Data comes in arrays/matrices (features, samples)
- operations like gradient descent forward propagation, etc. require massive matrix operations
- vectorization allows you to run models fast enough for training and experimentation.

# Writing custom Aggregate function pandas

Creating a custom aggregate function in pandas is powerful way to perform specialized operation during group-by or aggregation tasks.

Ex1: Custom aggregation with apply.

```
import pandas as pd

data = {'category': ['A', 'A', 'B', 'B', 'C'],
        'values': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)

def custom_range(series):
    return series.max() - series.min()

result = df.groupby('category')['values'].apply(custom_range)
print(result)
```

Ex2: Using agg with Named Custom functions.

```
def custom_mean_plus_10(series):
    return series.mean() + 10

result = df.groupby('category').agg({'values': custom_mean_plus_10})
print(result)
```

Ex 3:- Multiple Aggregations with Custom and Built-in function.

```
aggregations = {
    'values': [custom_range, 'sum', 'mean']
}

result = df.groupby('category').agg(aggregations)
print(result)
```