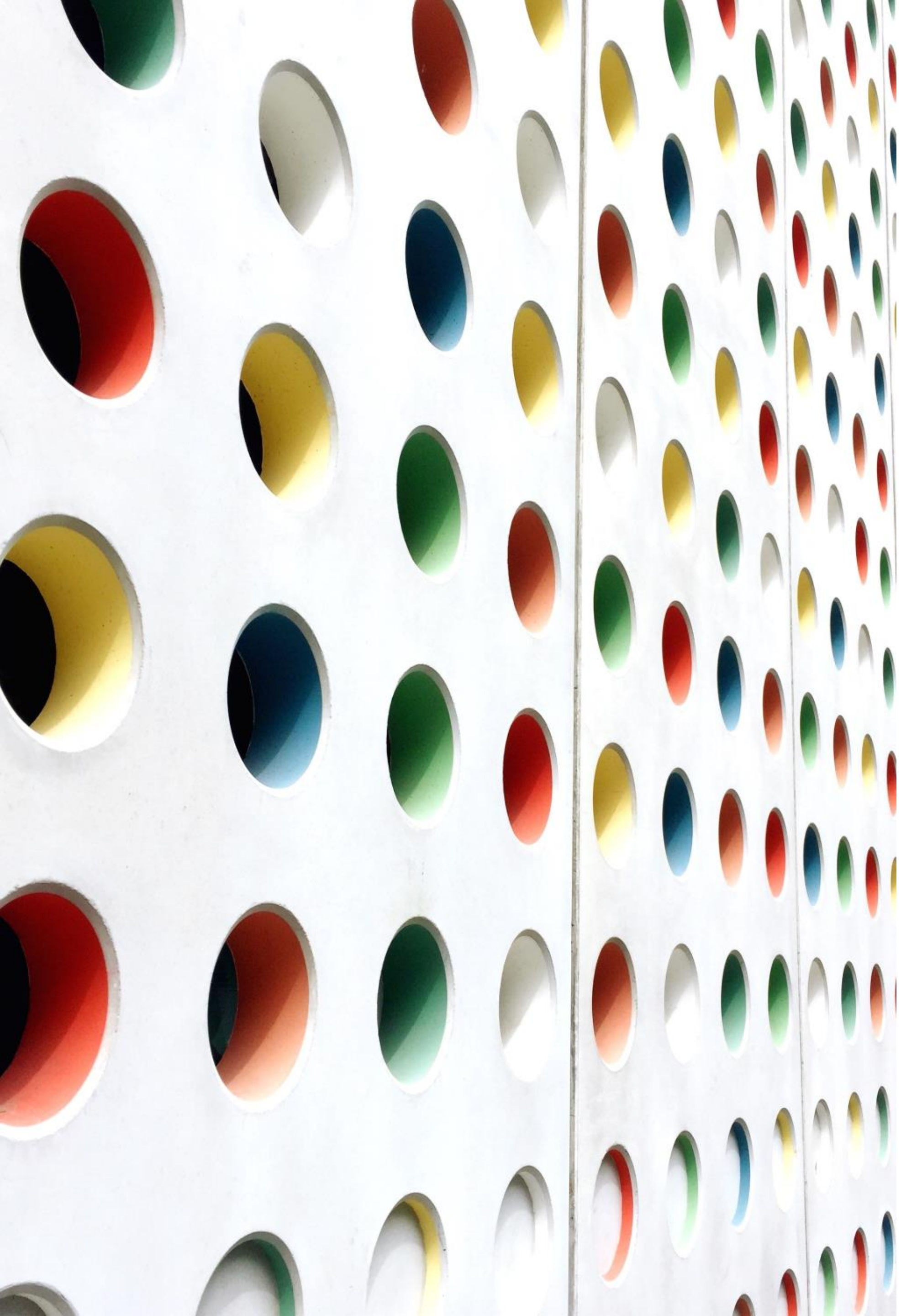


Pandas Cheat Sheet

Deepali Srivastava

Author of [“Ultimate Python Programming”]

Access the Pandas Jupyter Notebook with 108 questions here 👇
https://github.com/Deepali-Srivastava/Pandas-Cheat-Sheet-for-Data-Analysis

Cheat Sheet Contents

Importing numpy and pandas

```
import numpy as np
import pandas as pd
```

Reading and writing data to files

```
df = pd.read_csv('data.csv')
df.to_csv('data.csv')
df = pd.read_excel('data.xlsx')
df.to_excel('data.xlsx')
```

Reading a CSV with specific options

```
df = pd.read_csv( 'data.csv',
                  sep=':', # CSV with colons as separating values
                  index_col=0, # Set the first column as the index
                  usecols=['Name', 'Age'], # Read only 'Name' and 'Age' columns
                  nrows=100, # Read first 100 rows only
                  dtype={'Age': float}, # Ensure 'Age' is read as a float
                  parse_dates=['column_2'], # Parse 'column_2' as datetime
                  encoding='utf-8' # Use UTF-8 encoding )
```

Reading a file that does not contain a header row

```
df = pd.read_csv('data.csv',
                 header=None, # First row not treated as header
                 names=['Name','Age','Phone'] # Manually specify column names
                )
```

Treat specific values as missing values(NaN) while reading a file

```
# Values 'n.a.', 'n/a' or 'null' in any column are treated as missing values
df = pd.read_csv('data.csv', na_values=['n.a.', 'n/a', 'null'])

# Column-Specific Custom Missing Values
df = pd.read_csv('data.csv', na_values={'email': ['unknown'], 'age':[-1], 'location': ['not available','n/a']})
```

Data Exploration

```
df.head(n) # Display the first n rows of the DataFrame
df.tail(n) # Display the last n rows of the DataFrame
df.sample(n) # Returns n random rows from the DataFrame
df.sample(frac=0.05) # Returns 5% random rows from the DataFrame
df.info() # Get a summary of the DataFrame, including data types and non-null values
df.describe() # Generate summary statistics for numerical columns
df.shape  # Get the dimensions of the DataFrame in a tuple (rows, columns)
df.size # Returns the total number of elements in the DataFrame (rows × columns)
df.columns # List all column names in the DataFrame
```

Exploring and converting data types

```
df.dtypes  # Get the data types of each column
df['age'].dtype # Get the data type of a specific column
df['salary'].astype('float64') # Convert the column from its existing data type into a float64
```

Indexing

```
# By default, Pandas assigns an integer index (0, 1, 2, 3, ...) to each row
df.index # Shows the row labels, which can be numeric (default) or custom labels (like strings, datetime, etc.)
df = df.set_index('name')  # Set 'name' column as the index
df =  df.reset_index() # Reset the index to the default integer index, original index will be added as a new column
'John' in df.index # Checking if a value exists in the index, Returns True or False
```

Correlation

```
# Getting pairwise correlation of numeric columns
correlation_matrix = df.corr(numeric_only=True) # Ensures only numeric columns are used

# Default method is 'pearson', can use 'kendall' or 'spearman’
correlation_matrix = df.corr(numeric_only=True ,method='pearson')

# Access a specific correlation value between two columns using .loc
correlation_matrix.loc['age','salary'] # Correlation between column 'age' and 'salary'
```

Commonly used aggregation functions

```
df.max() # Maximum value in each column
df['age'].max() # Maximum value in the 'age' column
df[['age','height']].max() # Maximum value in the 'age' and 'height' columns
df['age'].min() # Minimum value of 'age' column
df['age'].idxmax() # Index of the maximum value in 'age' column
df['age'].idxmin() # Index of minimum value in 'age' column
df.iloc[df['age'].idxmax()] # To get the row with maximum value
df['age'].mean() # Mean of the 'age' column
df['age'].median() # Median of the 'age' column
df['age'].mode() # returns most frequent value(s) of 'age' column in a Series
mode_value = df['age'].mode()[0] # Accesses the first mode value from the Series
df['age'].sum() # Sum of the 'age' column
df['age'].count() # Number of non-null values in the 'age' column
df['age'].std() # Standard deviation of the 'age' column
df['age'].var() # Variance of the 'age' column
df['age'].prod() # Product of all values in the 'age' column
df['age'].quantile(0.25) # 25th percentile(first quartile) of 'age' column
df['age'].describe() # Summary statistics for the 'age' column
df['age'].cumsum() # Cumulative sum of 'age' column

# Use axis=1 to perform operations on rows, so you get results like row sums, row means, etc.
df.min(axis=1) # Minimum value in each row
df.sum(axis=1) # Sum of each row
```

Iterating over a DataFrame

```
for index, row in df.iterrows():
    print(index) # Print the index of the current row.
    print(row) # Print the data of the current row.
```

Getting Unique Values and their Frequencies

```
# unique()returns an array of the unique values in the 'Class' column of df
df['Class'].unique()

# nunique() returns the number of unique values in the 'Class' column of df
df['Class'].nunique()

# Get a count per category for categorical columns
# value_counts() returns a Series containing the count of occurrences of each unique value
df['Class'].value_counts()   # Sorted in descending order by default, showing the most frequent values first

# Get the count of a specific category
count_grade_9 = df['Class'].value_counts()['Grade 9'] # Get the count of 'Grade 9' in 'Class' column
```

Identifying and Removing Duplicate Rows in a DataFrame

```
# Return a Boolean Series where True indicates a duplicate row.
df.duplicated()

# Count the number of duplicate (True) and unique (False) rows.
df.duplicated().value_counts()

# Remove all duplicate rows, keeping only the first occurrence.
df.drop_duplicates()  # By default, it considers all columns
df.drop_duplicates(subset=['column1', 'column2'])  # check for duplicates only in specific columns
```

Identifying and Removing Duplicate Columns in a DataFrame

```
df.T.duplicated()  # Check for duplicate columns by transposing the DataFrame and applying duplicated()
df.T.duplicated().value_counts() # Count how many columns are unique (False) and how many are duplicates (True)
df = df.T.drop_duplicates().T  # Remove duplicate columns
```

Data Selection

```
# Selecting columns
df['column'] # Select a single column as a Series
df[['column']] # Select a single column as a DataFrame
df[['col1', 'col2']] # Select multiple columns as a DataFrame

# Index based selection using iloc operator,.
df.iloc[row_positions, column_positions] # Select data based on its numerical position in the dataframe
df.iloc[0]  # Select a single row by position (here, the 0th row)
df.iloc[[0,2]] # Select multiple rows by position (here, the 0th and 2nd rows)
df.iloc[0,2]   # Select specific rows and columns, (here,the value at 0th row and 2nd column)

# Slicing in iloc, start is included, end is excluded
df.iloc[0:2] # Slice rows from position 0 to 2 (excluding position 2)
df.iloc[:, 0:2] # Slice columns from position 0 to 2 (excluding position 2)
df.iloc[1:3, 0:2]  #  Rows 1 to 3 (excluding 3) and columns 0 to 2 (excluding 2)
df.iloc[:, 1] # Column at position 1

# Label based selection using loc operator,
df.loc[row_label, col_label] # Access rows and columns by label
df.loc['a'] # Select a single row by label
df.loc[['a', 'c']]  # Select multiple rows by label
df.loc['a', 'City'] # Select a specific row and column
df.loc[['a', 'b'], ['Name', 'Age']]  # Select a subset of rows and columns
df.loc[df['Age'] > 30] # Boolean indexing to filter rows based on a condition

# Slicing in loc: both start and end are included
df.loc['b':'d', 'Name':'City'] # Rows from 'b' to 'd' and columns from 'Name' to 'City'
df.loc['a':'c'] # Slice rows from 'a' to 'c'
df.loc[:, 'Name':'Age'] # Slice columns from 'Name' to 'Age'
df.loc['b':'d', 'Age'] # Rows from 'b' to 'd', only 'Age' column
df.loc[df['Age'] > 30, 'Name':'City'] # Select rows where 'Age' > 30 and specific columns
df.loc[df['Age'] > 25, 'Name']   # Slice rows with a condition and only 'Name' column
```

Conditional Data selection: Filter rows based on a condition or multiple conditions

```
# Use relational operators (<, >, ==, >=, <=, !=) to filter rows
df[df['age'] > 10]

# Combine multiple conditions using logical operators   &   |   ~
df[(df['age'] > 20) & (df['score'] > 85)]   # Each condition should be enclosed in parentheses
df[(df['age'] > 10) | (df['height'] <150)]

# To filter rows where a column's value is in a specified list, use .isin()
df[df['city'].isin(['Bangalore', 'Bareilly', 'Agra'])]
df[~df['city'].isin(['Bangalore', 'Bareilly', 'Agra'])]

# Use .between() to filter rows where a column's values fall within a specified range
df[df['age'].between(20, 30)] # both 20 and 30 inclusive
df[df['age'].between(20, 30, inclusive = 'both')] # default - both 20 and 30 inclusive
df[df['age'].between(20, 30, inclusive='neither')] # Exclude both 20 and 30 from the filter
df[df['age'].between(20, 30, inclusive='left')] # Include 20 but exclude 30
df[df['age'].between(20, 30, inclusive='right')] # Exclude 20 but include 30

# Selecting specific columns after filtering
df[ cond1 & cond2][['col1', 'col2', 'col3']][:5]

# To filter rows where a column contains a specific string, use .str.contains()
df[df['Name'].str.contains('ux')]
```

Changing values using .loc

```
# Modify a single value
df.loc[1, 'age'] = 32  # Change 'age' of row with index 1(using integer index)
df.loc['Jim', 'age'] = 32  # Change 'age' of row with index 'Jim'( if 'name' column set as index)
# Modify multiple values
df.loc[0:1, 'city'] = 'Bengaluru' # Update 'city' for rows with index 0 and 1
# Modify based on a condition
df.loc[df['age'] > 30, 'age'] += 5  # Increase 'age' by 5 for rows where 'age' > 30
df.loc[df['gpa'] < 2.5, 'result'] = 'fail'    # Assign 'fail' to the 'result' column for rows where 'gpa' < 2.5
# Update an entire row
df.loc[1] = ['N/A', 'N/A', 'N/A']    # Set all values for row with index 1 to 'N/A'
# Update an entire column
df.loc[:,'city'] = 'Bareilly' # Set all values for 'city’ column
```

Changing values using .iloc

```
# Modify a Single Value
df.iloc[0, 2] = 'Agra'   # Change value at 0th row and 2nd column
# Modify Multiple Values
df.iloc[0:2, 1] = [28, 35]  # For the 0th and 1st rows, Change 1st column values
# Update an entire row
df.iloc[1] = ['Ram', 32, 'Bareilly'] # Change the 1st row
# Update an entire column
df.iloc[:, 2] = 'Lucknow' # Set all values for 2nd column to 'Lucknow'
```

Faster single value updates using .at(label-based) or .iat(index-based)

```
df.at[1, 'name'] = 'Maruti' # Change 'name' of the row with index 1 to 'Maruti'
df.iat[1, 1] = 36 # Update 1st column of 1st row
```

Replacing Values

```
df = df.replace('Yes', 'Y')     # Replace a single value in the entire DataFrame
df = df['result'].replace('Pass','P')    # Replace a single value in a specific column
df['result'] = df['result'].replace(['Pass', 'Fail'],['P', 'F']) # Replace multiple values in a column
df['result'] = df['result'].map({'Pass':'P',  'Fail':'F'}) # Replace multiple values using map
df['result'] = df['result'].replace(['Pass', 'Good', 'Satisfactory'], '1') # Replace multiple values with a single value

# Replacing multiple values in multiple columns
df = df.replace({
    'result': ['Pass', 'Fail'],
    'grade': ['A', 'B', 'C', 'D', 'F']
}, {
    'result': ['P', 'F'],
    'grade': ['4', '3', '2', '1', '0']
})
```

Clipping : Limiting the values within a specified range(useful in handling outliers)

```
# Clip values in column 'A' to be between 15 and 40
df['A'] = df['A'].clip(lower=15, upper=40) # Values lower than 15 replaced by 15, Values greater than 40 replaced by 40
# Clip all values in entire DataFrame to be between 10 and 30
df_clipped = df.clip(lower=10, upper=30)
# Clip column 'A' to be between 15 and 40, and column 'B' to be between 10 and 40
df_clipped = df.clip(lower={'A': 15, 'B': 10}, upper={'A': 40, 'B': 40})
```

Adding or modifying columns

```
# Add a new column or modify if the column already exists
df['country'] = 'India'  # Creates a column with the same value for all rows (or updates if column exists)
df['age_in_months'] = df['age'] * 12  # Creates a column using an existing column (or updates if column exists)
df['total'] = df['marksA'] + df['marksB'] # Using a Calculation with multiple Columns
df['Maximum Marks'] =  df['Maximum Marks'] + 10  # Modify the column
# Vectorized operation to add/modify a column based on 2 columns
df['type'] = np.where((df['age'] < 15) & (df['medals'] > 5), 'exceptional', 'normal')
```

Apply a function to add/modify a column

```
df['subject'] =df['subject'].apply(lambda name :name.strip().lower()) # Strip and lowercase text

def age_category(age):
   return 'Young' if age < 18 else 'Adult'
df['category'] = df['age'].apply(age_category) # Apply age_category function to 'age' column


def func(age, medals):
    if age < 15 and medals> 5:
        return 'exceptional'
    else:
        return 'normal'
# Apply np.vectorize to the function for vectorized operation
df['type'] = np.vectorize(func)(df['age'], df['medals'])
```

Removing columns

```
df = df.drop('name', axis=1)    # Drop the 'name' column
df = df[['name','age','phone']]  # Keep only 'name', 'age', and 'phone' columns, other columns dropped
```

Removing rows based on label index

```
df = df.drop('134ABC',errors='ignore') # Avoid error if index doesn't exist
```

Changing column names

```
df = df.rename(columns={'col1':'new_col1','col2':'new_col2', 'col3':'new_col3'}) # Rename specific columns
df.columns = ['new_col1', 'new_col2', 'new_col3'] # Rename all columns at once(make sure the length matches)
df.columns = [col.replace('%', '') for col in df.columns] # Remove '%' from column names
```

Finding missing Values

```
df.isnull()    # Returns a DataFrame with True for missing values
df.notnull()   # Detect non-missing values
df.isnull().sum() # Count Missing Values in Each Column
df.isnull().values.any() # Check if Any Missing Value Exists
df[df.isnull().any(axis=1)] # Locate Rows with Missing Values
```

Filling missing values

```
# Fill all missing values in the DataFrame with 'unknown'
df = df.fillna('unknown')

# Fill all missing values in column 'gender' with 'unknown'
df['gender'] = df['gender'].fillna('unknown')

# Fill missing values in different columns with different values
df = df.fillna({'gender': 'unknown', 'age': -1, 'country': 'India'})

# Fill missing values in a column with column mean/median/mode
df['column_name'] = df['column_name'].fillna(df['column_name'].mean())
df['column_name'] = df['column_name'].fillna(df['column_name'].median())
df['column_name'] = df['column_name'].fillna(df['column_name'].mode()[0])

# Fill Missing values with the Previous/Next Value
df = df.fillna(method='ffill') # Forward fill (uses previous row's value)
df = df.fillna(method='bfill') # Backward fill (uses next row's value)
# Fill missing values with interpolation
df['column_name'] = df['column_name'].interpolate() # Interpolate between available values
```

Removing rows/columns that contain missing values

```
# Drop rows with any missing values
df = df.dropna()

# Drop rows with all values missing
df = df.dropna(how='all')

# Drop rows where 'price' and 'quantity' columns have missing values
df = df.dropna(subset=['price', 'quantity'])

# Drop rows that have fewer than 4 non-missing values, keeps rows where at least 4 columns have valid data
df = df.dropna(thresh=4) # thresh denotes minimum number of non-NaN values

# Drop columns with any missing values
df = df.dropna(axis=1)

# Drop any columns that have fewer than 100 non-null values in them
df = df.dropna(axis=1, thresh=100)
```

Copying a DataFrame or a part of it

```
# Assigning a DataFrame or part of it creates reference, modifying one will affect the other
df_1 = df
df_adults = df[df['age'] > 18]

# Create an independent copy using copy()
# Copying entire DataFrame
df_copy = df.copy()

# Copying Rows
some_rows = df.iloc[:2].copy()   #  Copy the first two rows
df_adults = df[df['age'] > 18].copy()   # Copy rows where Age > 30

# Copying Columns
df_1 =  df[['name', 'city']].copy()   # Copy only the 'name' and 'city' columns
```

Sorting values in columns

```
# Sort by single column, default is ascending order
df = df.sort_values('column_name')
df = df.sort_values(by='column_name') # Using the 'by' parameter for better clarity

# Sort by single column in descending order
df = df.sort_values(by='column_name', ascending = False)

# Sort by col1(ascending order) then by col2(descending order)
df = df.sort_values(by=['col1', 'col2'], ascending=[True, False])

# Sort Rows by Index, default is ascending order
df.sort_index()

# Sort Rows by Index in descending order
df.sort_index(ascending=False)

# Sort column names, default is ascending order
df.sort_index(axis=1)

# Descending Column Names
df.sort_index(axis=1, ascending=False)

# key parameter for custom sorting
df.sort_values(by='column_name', key=lambda col: col.str.len())  # Sorting strings by length

# Filter the rows based on condition, select specific columns and then sort on columns 'col2'
df[condition][['col1', 'col2', 'col3']].sort_values('col2')
```

Getting n largest and n smallest values for a column

```
# Using nsmallest and nlargest() is more efficient than using sort_values with head() or tail()
df.nlargest(3, 'Marks') # Get top 3 largest values for 'Marks' column
df.nsmallest(3, 'Marks') # Get top 3 smallest values for 'Marks' column
```

concat() : Combine DataFrames either vertically (stacking rows) or horizontally (joining columns)

```
# Concatenate vertically (stack rows).
# If DataFrames have different columns, missing columns are filled with Nan values
df_vertical = pd.concat([df1, df2], axis=0, ignore_index=True) # ignore_index=True resets the index after concatenation

# Concatenate horizontally (join columns)
#If indices do not match for some rows, it will result in NaN values for the rows that don't exist in one of the DataFrames
df_horizontal = pd.concat([df1, df2], axis=1)  # rows are aligned by their index

# Concatenate with keys to distinguish the DataFrames, Adds hierarchical indexing
df_all= pd.concat([males_df, females_df], axis=0, ignore_index=True, keys=['males', 'females'])
df_all.loc['males']   # Access data by key
df_all.loc['females'] # Access data by key
```

merge(): Combine two DataFrames based on common columns or indices

```
pd.merge(df1, df2, on='ID', how='inner')
pd.merge(df1, df2, on='ID') # by default inner merge
pd.merge(df1, df2, on='ID', how='left')
pd.merge(df1, df2, on='ID', how='right')
pd.merge(df1, df2, on='ID', how='outer')
```

inner left right outer

```
# Joining with different column names in the two DataFrames
pd.merge(df1, df2, left_on='ID', right_on='EmpID')

# Joining on column in left DataFrame and index in right DataFrame
pd.merge(df1, df2, left_on='ID', right_index=True)

# Merge with indicator flag, adds a new column to the resulting DataFrame called _merge
# _merge column indicates whether each row comes from left only, right only or both DataFrames
pd.merge(df1, df2, on='ID', how='outer', indicator=True)

# Merge with custom suffixes
# By default _x and _y added to columns with same names in both DataFrames(except column(s) used to join)
pd.merge(df1, df2, on='ID', how='outer', suffixes=('_df1', '_df2')) #Adds suufixes to the overlapping column names
```

Grouping and aggregating data : Analysing data per category

Common aggregations

```python
# Group by 'column_name' and apply aggregation functions
df.groupby('column_name').sum() # Sum of numeric columns
df.groupby('column_name').mean() # Mean of numeric columns
df.groupby('column_name').count() # Count of non-null values
df.groupby('column_name').min() # Minimum value for each group
df.groupby('column_name').max() # Maximum value for each group
df.groupby('column_name').describe() # Summary statistics for each group
df.groupby('column_name').plot() # Plot grouped data

df.groupby('grade').mean()     # Group by 'grade'column and apply mean() for all numeric columns
df.groupby('grade')['height'].mean()    # Group by 'grade'column and apply mean() only for 'height' column
df.groupby('grade')[['height', 'age']].mean()   # Group by 'grade'column and apply mean() for 'height' and 'age' columns
```

Multiple aggregations using agg()

```python
# Apply min(), mean() and max() for 'height' and 'age' columns
df.groupby('grade')[['height', 'age']].mean().agg(['min', 'mean', 'max'])

# Apply max() for 'age' column and mean() for 'height' column
df.groupby('grade').agg({ 'age': 'max', 'height': 'mean' })
```

Custom aggregation

```python
df.groupby('grade').agg(func)     # Apply custom function func()
df.groupby('store').agg({ 'sales': ['sum', 'mean'],
                          'items_sold': lambda x: x.sum() / len(x) })
```

Multilevel grouping - grouping data by multiple columns

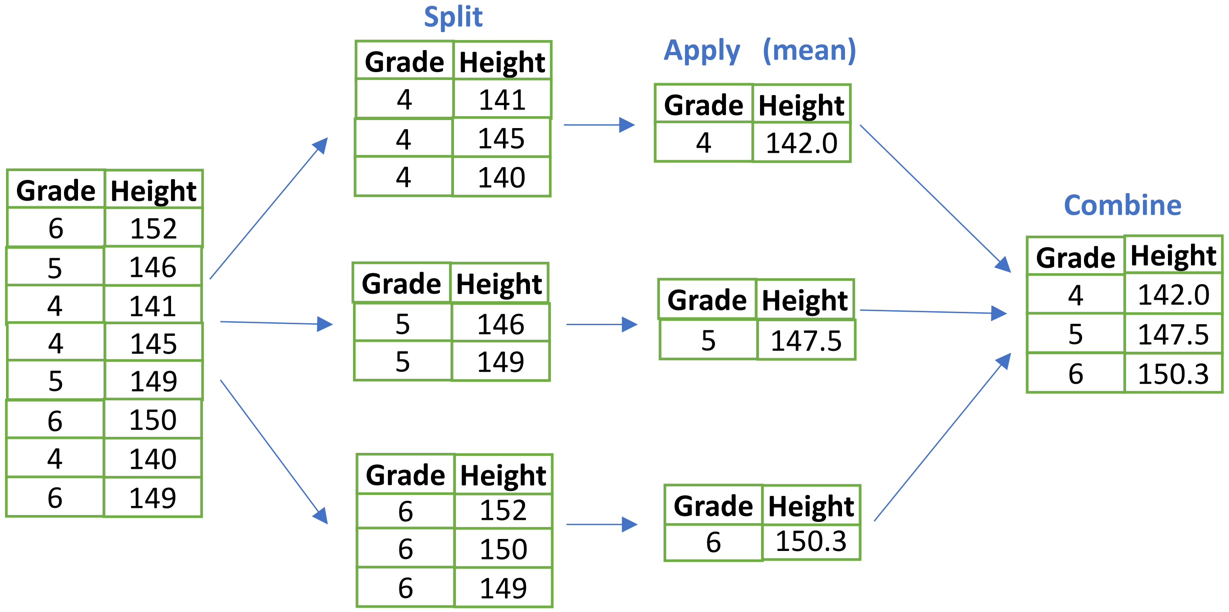
```python
df.groupby(['grade', 'section']).mean()    # grouped first by 'grade' then by 'section'
df.groupby(['grade', 'section']).mean().reset_index() # convert the hierarchical index into a flat table
```

Groupby on filtered dataset

```python
df[['name','age','grade']].groupby('grade').mean()
df[df['age'] > 10].groupby('grade').mean()
df[df['grade'].isin(['4','6'])].groupby('grade').mean()
```

```
# After grouping, the grouped column(s) become the index,to reset the index call reset_index()
df.groupby('column_name').sum().reset_index()

# Prevent the grouped column from becoming the index
df.groupby('column_name', as_index=False).sum() # as_index=False keeps 'column_name' as a regular column

# Include or exclude entire groups based on some condition
df.groupby('column_name').filter(func)  # Apply custom function 'func' to filter groups

# Include only those grades that contain more than 10 rows
df.groupby('grade').filter(lambda x : x.shape[0] > 10)

# Include only those grades for which average height is greater than 150
df.groupby('grade').filter(lambda x : x['height'].mean() > 150)

# Group by 'grade' column, find sum of numeric columns,sort the resulting DataFrame by the 'age' column
df.groupby('grade').sum().sort_values('age')
```

Grouping and aggregation works by split-apply-combine




```
# Finding average height of students in each grade
df.groupby('grade')['height'].mean()
```

Iterating through groups

```
g = df.groupby('grade')  # creates a GroupBy object by grouping the rows based on the unique values in the column 'grade'
for grade, data in g:
    print('Grade -', grade) # Print the group name (grade)
    print('Data - ')
    print(data) # Print the data for the specific group
    print('\n')
```

```
# Apply aggregate functions on the GroupBy object
g.mean() # Returns the mean value for each column in each group
g.max() # Returns the maximum value for each column in each group
g.size() # Returns the size of each group (number of rows per group)
```

Retrieve the DataFrame for a specific group

```
g = df.groupby('grade')
g.get_group('6')
df.groupby('column_name').get_group('group_value')
```

```
# Group by 'region' and 'store', get data for region='East' and store='A'
df.groupby(['region', 'store']).get_group(('East', 'A'))
```

Working with strings

```
# .str accessor used to apply string methods to columns
df['name_stripped'] = df['name_with_spaces'].str.strip() # Remove leading and trailing spaces
df['name_replaced'] = df['name'].str.replace('a', '@') # Replace 'a' with '@' in the 'name' column
df['country'].str.upper().value_counts() # Convert 'country' column to uppercase and get value counts

df['name_split'] = df['name'].str.split(' ') # Split the 'name' column into a list of words
df['first_name'] = df['name'].str.split(' ').str[0] # Get the first name (first element of list)

# Split the 'name' column into first and last name and expand into separate columns
df[['first_name', 'last_name']] = df['name'].str.split(' ', expand=True) # Expand into two new columns
```

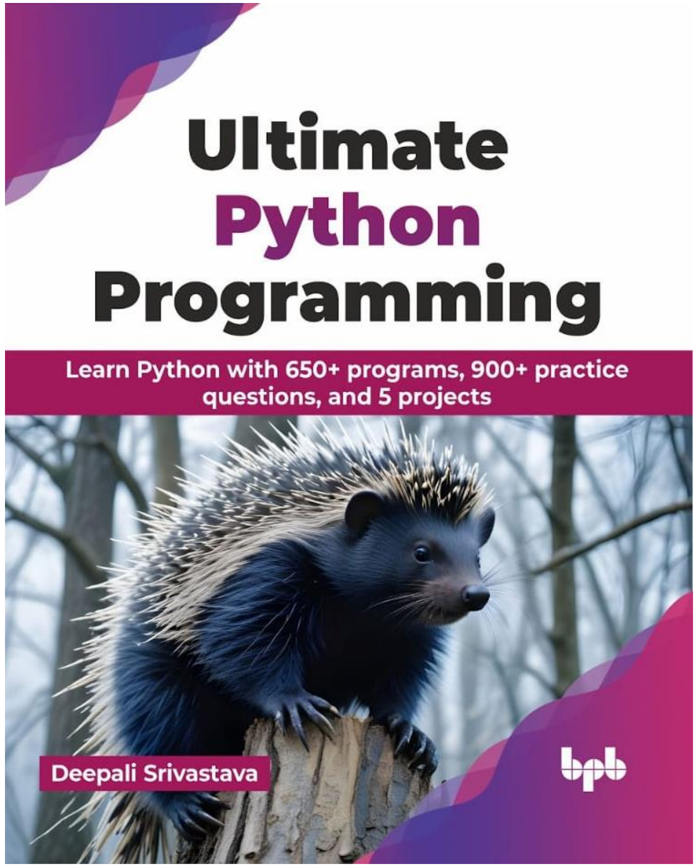
Working with datetime

```
# Converting a Column to Datetime
df['dob'] = pd.to_datetime(df['dob']) # Convert the 'dob' column to datetime

# Extract various components (year, month, day) from a datetime column.
df['birth_year'] = df['dob'].dt.year # Extract year from 'dob'
df['birth_month'] = df['dob'].dt.month # Extract month from 'dob'
df['birth_day'] = df['dob'].dt.day # Extract day from 'dob'
df['dob'].dt.weekday # Get weekday (integer: Monday=0, Sunday=6)
df['dob'].dt.day_name() # Get the name of the day (e.g., 'Monday', 'Tuesday')

# Convert a datetime object back to a string with a specific format using strftime
df['formatted_date'] = df['date'].dt.strftime('%Y-%m-%d') #  Format datetime as 'YYYY-MM-DD'

# Calculate the difference between two datetime columns using subtraction which results in aTimedelta object
df['duration'] = df['end_date'] - df['start_date'] # Time difference between 'end_date' and 'start_date'
df['days'] = df['duration'].dt.days  # Extract the number of days from the Timedelta

# Filtering by Date
df[df['date'] > '2025-01-01'] # Filter rows where 'dob' is after January 1, 2025
```

Inserting Missing Dates

```
# Create a complete date range from January 1, 2024, to January 8, 2024  with daily frequency
daterange = pd.date_range('2024-01-01', '2024-01-08', freq='D')

# Ensure the 'date' column is a datetime type and set it as index
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)

# Reindex the DataFrame df to the new date range, Missing dates will create new rows with NaN values in all columns
df = df.reindex(daterange)
```

Learn Python with 650+ Programs, 900+ Practice Questions, and 5 Projects



Access the Pandas Jupyter Notebook with 108 questions here 👇
https://github.com/Deepali-Srivastava/Pandas-Cheat-Sheet-for-Data-Analysis