No.	Category	Subcategory	Function Name	Imp_Arguments	Function Description	Example
1	DataFrame and Series Creation	Creating DataFrames	*pd.DataFrame()*	data: Dict, list, or DataFrame Data to populate the DataFrame columns: List of column names index: List of row labels (optional).	The `pd.DataFrame()` function is used to create DataFrame objects.	df = pd.DataFrame(data)
2	DataFrame and Series Creation	Creating Series	*pd.Series()*	data: List, ndarray, or scalar index: List of labels for the Series dtype: Data type of the resulting Series.	The `pd.Series()` function is used to create a Series object.	series = pd.Series([1, 2, 3, 4, 5])
3	DataFrame and Series Creation	Reading CSV Files	*pd.read_csv()*	filepath_or_buffer: File path or URL sep: Delimiter (default is ',') header: Row number to use as column names (default is 0) index_col: Column to set as the index (optional) usecols: Subset of columns to read (optional) dtype: Data types for the columns (optional).	The `pd.read_csv()` function reads CSV files into DataFrame objects.	df_csv = pd.read_csv('example.csv')
4	DataFrame and Series Creation	Reading Excel Files	*pd.read_excel()*	io: File path or file-like object sheet_name: Name or index of sheet to read (default is 0) header: Row to use for column headers index_col: Column(s) to set as index.	The `pd.read_excel()` function reads Excel files into DataFrame objects.	df_excel = pd.read_excel('example.xlsx')
5	DataFrame and Series Creation	Reading JSON Files	*pd.read_json()*	path_or_buffer: File path or JSON string orient: The format of the JSON string (default is 'records') dtype: Data types for columns.	The `pd.read_json()` function reads JSON data into DataFrame objects.	df_json = pd.read_json('example.json')
6	DataFrame and Series Creation	Concatenating DataFrames	*pd.concat()*	objs: List of DataFrames or Series axis: Axis along which to concatenate (0 for rows, 1 for columns) ignore_index: Whether to reset the index after concatenation (default is False) join: How to handle indices (default is 'outer').	The `pd.concat()` function is used to concatenate DataFrames along rows or columns.	df_concat = pd.concat([df1, df2], axis=0)
7	Data Selection and Indexing	Label-based Indexing	*.loc[]*	row/column labels: Specify rows and columns by labels columns: Columns to select (optional) slices: Slicing rows or columns (inclusive of both ends).	The `.loc[]` function is used to select rows and columns by labels.	selected_rows = df.loc[0:1], selected_columns = df.loc[:, 'Name']

8	Data Selection and Indexing	Integer-location based Indexing	*.iloc[]*	row/column indices: Specify rows and columns by their integer positions slices: Slicing rows or columns (exclusive of the end index).	The `.iloc[]` function is used to select rows and columns by their integer index.	`selected_rows_iloc = df.iloc[0:2], selected_columns_iloc = df.iloc[:, 0]`
9	Data Selection and Indexing	Fast Access to Single Elements	*.at[], .iat[]*	.at[]: Takes row and column labels .iat[]: Takes integer positions for row and column.	The `.at[]` and `.iat[]` functions are used to access single values by label or integer position, respectively.	`single_value_at = df.at[0, 'Age'], single_value_iat = df.iat[0, 1]`
10	Data Selection and Indexing	Querying the DataFrame	*.query()*	expr: A string expression to filter the DataFrame inplace: Whether to modify the DataFrame in place (optional).	The `.query()` function allows you to query a DataFrame using a string expression.	`df_query = df.query('Age > 30')`
11	Data Selection and Indexing	Setting the Index of a DataFrame	*.set_index()*	keys: Column name(s) to set as the index inplace: Whether to modify the DataFrame in place (optional) drop: Whether to drop the column after setting it as the index (default is True).	The `.set_index()` function allows you to set a column as the index.	`df_indexed = df.set_index('Name')`
12	Data Selection and Indexing	Resetting the Index	*.reset_index()*	level: Level(s) to reset (if multi-level index) drop: Whether to drop the index (default is False) inplace: Whether to modify the DataFrame in place (optional).	The `.reset_index()` function is used to reset the index of a DataFrame.	`df_reset = df_indexed.reset_index()`
13	Data Selection and Indexing	Selecting Data at Particular Level	*.xs()*	key: The value to select from a particular level level: The level to query (for multi-level index) axis: Axis along which to select (0 for rows, 1 for columns).	The `.xs()` function is used to get a cross-section from the DataFrame.	`df_xs = df.set_index(['City', 'Name']).xs('New York', level='City')`
14	Data Cleaning	Dropping Missing Values	*.dropna()*	axis: Axis along which to drop (0 for rows, 1 for columns) how: Drop rows/columns where all or any values are NaN ('any' or 'all') thresh: Require a minimum number of non-NaN values subset: Specify subset of columns to check for missing values.	The `.dropna()` function is used to remove missing values (NaN).	`df_cleaned = df.dropna()`

15	Data Cleaning	Filling Missing Values	*.fillna()*	value: Value to fill NaN values with method: Method to fill NaNs ('pad' for forward fill, 'bfill' for backward fill) axis: Axis along which to fill limit: Maximum number of replacements.	The `.fillna()` function is used to fill missing values with a specified value.	`df_filled = df.fillna({'Age': 0, 'City': 'Unknown'})`
16	Data Cleaning	Replacing Values	*.replace()*	to_replace: The value or dictionary to replace value: The value to replace with regex: Whether to interpret 'to_replace' as a regular expression (optional).	The `.replace()` function is used to replace specific values in the DataFrame.	`df_replaced = df.replace({'Age': {30: 32}})`
17	Data Cleaning	Checking for Missing Values	*.isna() and .notna()*	None Returns a DataFrame of the same shape with True for NaN values and False for others.	The `.isna()` function returns a boolean mask indicating where NaN values are present. The `.notna()` function returns the opposite mask.	`df_isna = df.isna(), df_notna = df.notna()`
18	Data Cleaning	Detecting Duplicates	*.duplicated()*	subset: Columns to check for duplicates keep: Which duplicates to mark as True ('first', 'last', or False).	The `.duplicated()` function is used to find duplicate rows in the DataFrame.	`df_duplicates = df.duplicated()`
19	Data Cleaning	Dropping Duplicate Rows	*.drop_duplicates()*	subset: Columns to check for duplicates keep: Which duplicates to retain ('first', 'last', or False) inplace: Whether to modify the DataFrame in place (optional).	The `.drop_duplicates()` function is used to remove duplicate rows from the DataFrame.	`df_no_duplicates = df.drop_duplicates()`
20	Data Transformation	Applying Functions to Data	*.apply()*	func: Function to apply axis: Axis along which to apply the function (0 for rows, 1 for columns) raw: Whether to pass the underlying data as a raw ndarray.	The `.apply()` function is used to apply a function along the axis of the DataFrame or Series.	`df_applied = df.apply(lambda x: x.max() - x.min())`
21	Data Transformation	Mapping Functions to Series	*.map()*	arg: A function, dictionary, or Series to map na_action: Action to take for missing values ('ignore' or None).	The `.map()` function is used to map values from one set to another.	`df_mapped = df['Age'].map({25: 'Young', 30: 'Middle-aged', 35: 'Old'})`
22	Data Transformation	Element-wise Function Application (for DataFrame)	*.applymap()*	func: Function to apply element-wise na_action: Action to take for missing values ('ignore' or None).	The `.applymap()` function applies a function element-wise to the DataFrame.	`df_applymap = df.applymap(lambda x: len(str(x)))`

23	Data Transformation	Grouping Data	*.groupby()*	by: Column(s) or index level(s) to group by axis: Axis along which to group as_index: Whether to set the grouped column as the index (default is True).	The `.groupby()` function is used to group data by one or more columns, and then apply aggregation functions to the groups.	`df_grouped = df.groupby('City').mean()`
24	Data Transformation	Creating Pivot Tables	*.pivot_table()*	values: Column(s) to aggregate index: Column(s) to use as the index columns: Column(s) to use as columns aggfunc: Aggregation function (default is 'mean').	The `.pivot_table()` function is used to create pivot tables for summarizing data.	`df_pivot = df.pivot_table(values='Age', index='City', columns='Name', aggfunc='mean')`
25	Data Transformation	Reshaping Data (Wide to Long)	*.melt()*	id_vars: Columns to keep as identifiers value_vars: Columns to unpivot var_name: Name for the new 'variable' column value_name: Name for the new 'value' column.	The `.melt()` function is used to reshape data from wide format to long format.	`df_melted = pd.melt(df, id_vars=['City'], value_vars=['Age'])`
26	Data Transformation	Stack Data (Wide to Long)	*.stack()*	level: Level(s) to stack (for multi-level index) dropna: Whether to exclude missing values.	The `.stack()` function stacks the columns of the DataFrame into a Series.	`df_stacked = df.stack()`
27	Data Transformation	Unstack Data (Long to Wide)	*.unstack()*	level: Level(s) to unstack (for multi-level index) fill_value: Value to fill for missing values (optional).	The `.unstack()` function converts a stacked Series back into a DataFrame.	`df_unstacked = df_stacked.unstack()`
28	Merging, Joining, and Concatenating	Merging DataFrames	*.merge()*	right: DataFrame to merge with on: Column or index level to join on how: Type of merge ('left', 'right', 'outer', 'inner') left_on, right_on: Columns to join on (if they are not the same in both DataFrames) suffixes: Suffixes to apply to overlapping column names.	The `.merge()` function is used to merge two DataFrames on one or more columns.	`df_merged = pd.merge(df, df2, on='City', how='inner')`
29	Merging, Joining, and Concatenating	Joining DataFrames by Index	*.join()*	other: DataFrame to join with on: Column(s) to join on (optional) how: Type of join ('left', 'right', 'outer', 'inner') lsuffix, rsuffix: Suffixes to apply to overlapping columns.	The `.join()` function is used to join two DataFrames based on their index.	`df_joined = df.set_index('Name').join(df2.set_index('Name'))`

30	Merging, Joining, and Concatenating	Concatenating DataFrames	*.concat()*	objs: A sequence or mapping of DataFrames to concatenate axis: Axis along which to concatenate (0 for rows, 1 for columns) join: Type of join ('outer', 'inner') ignore_index: Whether to reset the index.	The `.concat()` function is used to concatenate multiple DataFrames along a particular axis (rows or columns).	df_concat = pd.concat([df, df2], axis=0, ignore_index=True)
31	Merging, Joining, and Concatenating	Appending DataFrames	*.append()*	other: DataFrame to append ignore_index: Whether to reset the index verify_integrity: Whether to check for duplicates.	The `.append()` function is used to append rows of one DataFrame to another.	df_appended = df.append(df2, ignore_index=True)
32	Merging, Joining, and Concatenating	Merge on Closest Key	*.merge_asof()*	right: DataFrame to merge with on: Column(s) to join on direction: Direction to match keys ('forward', 'backward', or 'nearest') by: Column(s) to group by.	The `.merge_asof()` function is used to merge DataFrames on the nearest key rather than exact matches.	df_asof = pd.merge_asof(df, df2, on='Date', direction='nearest')
33	Merging, Joining, and Concatenating	Merge Ordered DataFrames	*.merge_ordered()*	right: DataFrame to merge with on: Column(s) to join on fill_method: Method to fill missing values ('ffill', 'bfill') how: Type of merge ('left', 'right', 'outer', 'inner').	The `.merge_ordered()` function is used to merge two ordered DataFrames while preserving the order of rows.	df_ordered = pd.merge_ordered(df, df2, on='Date', fill_method='ffill')
34	Time Series	Converting to Datetime	*.to_datetime()*	arg: The argument to convert (e.g., a column or a list of strings) format: Optional format to infer datetime format errors: Action for invalid parsing ('raise', 'coerce', 'ignore').	The `.to_datetime()` function is used to convert a column or list to datetime objects.	df['Date'] = pd.to_datetime(df['Date'])
35	Time Series	Creating a Date Range	*.date_range()*	start: Starting date end: Ending date periods: Number of periods freq: Frequency (e.g., 'D', 'M', 'H', etc.) tz: Timezone.	The `.date_range()` function is used to create a range of dates with a specified frequency.	date_range = pd.date_range(start='2025-01-01', end='2025-12-31', freq='M')

36	Time Series	Resampling Time Series	*.resample()*	rule: Frequency rule (e.g., 'D' for daily, 'M' for monthly) how: Aggregation method (e.g., 'sum', 'mean', 'ohlc') label: Which side to label ('left' or 'right') closed: Which side to mark as closed ('left' or 'right').	The `.resample()` function is used to resample time series data to a different frequency.	`df_resampled = df.resample('M').mean()`
37	Time Series	Shifting Data	*.shift()*	periods: Number of periods to shift freq: Frequency to shift by (optional, for time-based data) axis: Axis along which to shift fill_value: Value to use for missing values after the shift.	The `.shift()` function shifts data by a specified number of periods.	`df_shifted = df['Value'].shift(1)`
38	Time Series	Rolling Window Calculations	*.rolling()*	window: Window size (e.g., number of periods) min_periods: Minimum number of periods required to compute a result axis: Axis along which to calculate the rolling statistics.	The `.rolling()` function provides a rolling view of the data to perform window-based calculations.	`df_rolling = df['Value'].rolling(window=3).mean()`
39	Time Series	Exponential Weighted Functions	*.ewm()*	span: The span of the exponential window min_periods: Minimum number of periods required to compute a result adjust: Whether to adjust the weights (default is True).	The `.ewm()` function provides an exponential weighted view of the data.	`df_ewm = df['Value'].ewm(span=3).mean()`
40	Handling Missing Data	Checking for Missing Values	*.isna()*	None	The `.isna()` function is used to detect missing values in a DataFrame or Series.	`df_isna = df.isna(), df_notna = df.notna()`
41	Handling Missing Data	Checking for Non-missing Values	*.notna()*	None	The `.notna()` function is used to detect non-missing values in a DataFrame or Series.	`df_notna = df.notna()`
42	Handling Missing Data	Dropping Missing Values	*.dropna()*	axis: Axis along which to drop missing values (0 for rows, 1 for columns) how: Method for dropping ('any' or 'all') thresh: Minimum number of non-NA values required to retain the row/column subset: Columns to consider for dropping.	The `.dropna()` function is used to drop missing values from a DataFrame or Series.	`df_dropped = df.dropna()`

43	Handling Missing Data	Filling Missing Values	*.fillna()*	value: Value to use for filling missing values (scalar, dict, or DataFrame) method: Method for filling ('ffill' for forward fill, 'bfill' for backward fill) axis: Axis along which to fill missing values (0 for rows, 1 for columns) limit: Maximum number of replacements to make.	The `.fillna()` function is used to fill missing values with a specified value or method.	`df_filled = df.fillna(0)`
44	Handling Missing Data	Interpolating Missing Values	*.interpolate()*	method: Method of interpolation ('linear', 'polynomial', etc.) axis: Axis along which to interpolate (0 for rows, 1 for columns) limit: Maximum number of missing values to fill inplace: Whether to modify the DataFrame in place.	The `.interpolate()` function is used to interpolate missing values in a DataFrame or Series.	`df_interpolated = df['Value'].interpolate()`
45	Handling Missing Data	Replacing Values	*.replace()*	to_replace: Value(s) to replace (single value, list, dict, or regex) value: Value(s) to replace with inplace: Whether to modify the DataFrame in place.	The `.replace()` function is used to replace specified values with others.	`df_replaced = df.replace({None: 0, 'N/A': -1})`
46	Grouping and Aggregating	Grouping Data	*.groupby()*	by: Column(s) to group by axis: Axis along which to group (default is 0, meaning rows) as_index: Whether to use group keys as the index (default is True) sort: Whether to sort the group keys (default is True).	The `.groupby()` function is used to group data by one or more columns.	`df_grouped = df.groupby('Category')['Value'].sum()`
47	Grouping and Aggregating	Aggregating Data	*.agg()*	func: Aggregation function(s) to apply (e.g., 'sum', 'mean', 'min') axis: Axis along which to apply the function (default is 0).	The `.agg()` function is used to apply aggregation functions to grouped data.	`df_agg = df.groupby('Category').agg({'Value': 'sum', 'Date': 'min'})`
48	Grouping and Aggregating	Transforming Grouped Data	*.transform()*	func: Function to apply to each group axis: Axis along which to apply the function (default is 0).	The `.transform()` function is used to transform grouped data while maintaining the original shape.	`df_transformed = df.groupby('Category')['Value'].transform(lambda x: (x - x.mean()) / x.std())`

49	Grouping and Aggregating	Applying Functions to Groups	*.apply()*	func: Function to apply to each group axis: Axis along which to apply the function (default is 0).	The `.apply()` function is used to apply a function along the group axis.	`df_apply = df.groupby('Category').apply(lambda x: x['Value'].max() - x['Value'].min())`
50	Grouping and Aggregating	Creating Pivot Tables	*.pivot_table()*	values: Column(s) to aggregate index: Column(s) to group by (rows of the pivot table) columns: Column(s) to group by (columns of the pivot table) aggfunc: Aggregation function (e.g., 'sum', 'mean') fill_value: Value to replace missing data with.	The `.pivot_table()` function is used to create pivot tables for aggregation.	`df_pivot = df.pivot_table(values='Value', index='Category', columns='Date', aggfunc='sum')`
51	Merging, Joining, and Concatenating	Merging DataFrames	*.merge()*	right: DataFrame to merge with on: Column(s) to join on how: Type of join ('left', 'right', 'outer', 'inner') left_on, right_on: Columns from left and right DataFrames to join on, if not the same.	The `.merge()` function is used to merge two DataFrames based on a key or index.	`df_merged = pd.merge(df1, df2, on='ID', how='inner')`
52	Merging, Joining, and Concatenating	Joining DataFrames	*.join()*	other: DataFrame to join on: Column(s) to join on (optional, defaults to index) how: Type of join ('left', 'right', 'outer', 'inner').	The `.join()` function is used to join two DataFrames on their index or a key column.	`df_joined = df1.set_index('ID').join(df2.set_index('ID'), how='outer')`
53	Merging, Joining, and Concatenating	Concatenating DataFrames	*.concat()*	objs: List of DataFrames to concatenate axis: Axis along which to concatenate (0 for rows, 1 for columns) ignore_index: Whether to ignore the index and create a new one join: Type of join ('inner' or 'outer') for columns.	The `.concat()` function is used to concatenate multiple DataFrames along a particular axis.	`df_concat = pd.concat([df1, df2], axis=0)`
54	Merging, Joining, and Concatenating	Merge on Closest Key	*.merge_asof()*	left, right: DataFrames to merge on: Column to merge on direction: Direction of merge ('forward', 'backward', 'nearest') by: Additional column(s) to group by.	The `.merge_asof()` function is used to merge two DataFrames on the nearest key.	`df_asof = pd.merge_asof(df1, df2, on='Date', direction='forward')`

55	Merging, Joining, and Concatenating	Merge with Ordered Data	*.merge_ordered()*	left, right: DataFrames to merge on: Column to merge on fill_method: Method to fill missing values ('ffill', 'bfill').	The `.merge_ordered()` function is used to merge DataFrames while maintaining the order of rows.	`df_ordered = pd.merge_ordered(df1, df2, on='Date', fill_method='ffill')`
56	Time Series	Converting to DateTime	*.to_datetime()*	arg: Date-like object to convert (e.g., string, list) format: String format to specify how dates are represented (optional) errors: How to handle errors ('raise', 'coerce', 'ignore').	The `.to_datetime()` function is used to convert a string or other data type to a datetime object.	`df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')`
57	Time Series	Resampling Time Series Data	*.resample()*	rule: Frequency for resampling (e.g., 'M' for monthly, 'W' for weekly) on: Column to use for time-based resampling (usually a datetime column) how: Aggregation method (e.g., 'sum', 'mean').	The `.resample()` function is used to resample time series data to a different frequency.	`df_resampled = df.resample('M', on='Date').sum()`
58	Time Series	Shifting Data	*.shift()*	periods: Number of periods to shift freq: Frequency to shift (optional) axis: Axis along which to shift (default is 0).	The `.shift()` function is used to shift the data in a Series or DataFrame by a specified number of periods.	`df_shifted = df['Value'].shift(1)`
59	Time Series	Rolling Window Calculations	*.rolling()*	window: Size of the rolling window min_periods: Minimum number of non-NA values required to compute the statistic center: Whether to set the label at the center of the window (default is False).	The `.rolling()` function is used to perform rolling window calculations.	`df_rolling = df['Value'].rolling(window=3).mean()`
60	Time Series	Creating a Date Range	*.date_range()*	start: Start date for the range end: End date for the range (optional) periods: Number of periods to generate (optional) freq: Frequency for the dates (e.g., 'D' for daily, 'M' for monthly).	The `.date_range()` function is used to create a range of dates with a specified frequency.	`date_range = pd.date_range(start='2020-01-01', periods=5, freq='D')`
61	Data Cleaning and Preprocessing	Detecting Missing Values	*.isnull()*	None Returns a DataFrame of the same shape as the original, with boolean values indicating missing data.	The `.isnull()` function is used to detect missing values in a DataFrame or Series.	`df_null = df.isnull()`

62	Data Cleaning and Preprocessing	Dropping Missing Values	*.dropna()*	axis: Axis to drop (0 for rows, 1 for columns) how: If 'any', drop rows/columns with any missing values If 'all', drop only rows/columns with all missing values thresh: Minimum number of non-NA values to keep.	The `.dropna()` function is used to drop rows or columns with missing values.	`df_dropped = df.dropna()`
63	Data Cleaning and Preprocessing	Filling Missing Values	*.fillna()*	value: Value to use for filling missing values method: Method to fill missing values ('ffill' for forward fill, 'bfill' for backward fill) axis: Axis along which to fill missing values.	The `.fillna()` function is used to fill missing values with a specified value or method.	`df_filled = df.fillna(0)`
64	Data Cleaning and Preprocessing	Replacing Values	*.replace()*	to_replace: Value(s) to replace value: Replacement value(s) inplace: Whether to modify the DataFrame in place (default is False).	The `.replace()` function is used to replace values in a DataFrame or Series.	`df_replaced = df.replace({'Category': {'A': 'Alpha', 'B': 'Beta'}})`
65	Data Cleaning and Preprocessing	Dropping Duplicate Rows	*.drop_duplicates()*	subset: Columns to consider for identifying duplicates (optional) keep: Which duplicates to keep ('first', 'last', or False to drop all) inplace: Whether to modify the DataFrame in place (default is False).	The `.drop_duplicates()` function is used to drop duplicate rows.	`df_unique = df.drop_duplicates()`
66	Data Cleaning and Preprocessing	Type Casting	*.astype()*	dtype: Data type to convert to (e.g., 'float', 'int', 'str') errors: How to handle errors ('raise', 'ignore', 'coerce').	The `.astype()` function is used to convert a column to a specific data type.	`df['Value'] = df['Value'].astype(float)`
67	Data Cleaning and Preprocessing	Applying Function to Entire DataFrame	*.applymap()*	func: Function to apply to each element of the DataFrame.	The `.applymap()` function is used to apply a function to each element of the DataFrame.	`df_squared = df.applymap(lambda x: x**2 if isinstance(x, (int, float)) else x)`

68	Text Data Handling	Checking for a Substring	*.str.contains()*	pat: Substring or regular expression to search for case: Whether to be case-sensitive (default is True) na: Value to return for missing values (default is NaN).	The `.str.contains()` function is used to check if a substring exists in each element of a Series.	`df['has_A'] = df['Category'].str.contains('A')`
69	Text Data Handling	Replacing Substring	*.str.replace()*	pat: Substring or regular expression to search for repl: Substring to replace with n: Maximum number of replacements (optional).	The `.str.replace()` function is used to replace a substring with another substring.	`df['Category'] = df['Category'].str.replace('A', 'Alpha')`
70	Text Data Handling	Splitting Strings	*.str.split()*	pat: Substring or regular expression to split by n: Maximum number of splits (optional).	The `.str.split()` function is used to split a string into multiple substrings.	`df['Category_split'] = df['Category'].str.split(' ')`
71	Text Data Handling	Converting to Lowercase	*.str.lower()*	None Converts all characters to lowercase.	The `.str.lower()` function is used to convert each string in a Series to lowercase.	`df['Category_lower'] = df['Category'].str.lower()`
72	Text Data Handling	Converting to Uppercase	*.str.upper()*	None Converts all characters to uppercase.	The `.str.upper()` function is used to convert each string in a Series to uppercase.	`df['Category_upper'] = df['Category'].str.upper()`
73	Text Data Handling	Stripping Leading and Trailing Whitespaces	*.str.strip()*	None Removes whitespaces from the beginning and end of each string.	The `.str.strip()` function is used to remove leading and trailing whitespaces from strings.	`df['Category_stripped'] = df['Category'].str.strip()`
74	Categorical Data Handling	Converting to Categorical Type	*.astype('category')*	dtype: Data type to convert to (in this case, 'category').	The `.astype('category')` function is used to convert a column to a categorical data type.	`df['Category'] = df['Category'].astype('category')`
75	Categorical Data Handling	Accessing Categorical Codes	*.cat.codes*	None Returns a Series of integers representing the categorical codes.	The `.cat.codes` attribute is used to access the integer codes for each category in a categorical column.	`df['Category_codes'] = df['Category'].cat.codes`
76	Categorical Data Handling	Accessing Categorical Categories	*.cat.categories*	None Returns a list of categories in the categorical column.	The `.cat.categories` attribute is used to get the list of categories in a categorical column.	`categories = df['Category'].cat.categories`

77	Categorical Data Handling	One-Hot Encoding	*.get_dummies()*	data: Column or DataFrame to encode prefix: Prefix for new column names drop_first: Whether to drop the first category to avoid multicollinearity (default is False).	The `.get_dummies()` function is used for one-hot encoding of categorical variables.	`df_encoded = pd.get_dummies(df['Category'])`
78	Categorical Data Handling	Grouping Data	*.groupby()*	by: Column(s) to group by axis: Axis along which to perform the operation (default is 0) as_index: Whether to set the group labels as the index (default is True).	The `.groupby()` function is used to group data based on categorical variables and perform aggregation.	`df_grouped = df.groupby('Category').sum()`
79	Categorical Data Handling	Creating Pivot Tables	*.pivot_table()*	values: Column(s) to aggregate index: Column(s) to group by aggfunc: Aggregation function (e.g., 'sum', 'mean').	The `.pivot_table()` function is used to create pivot tables from a DataFrame.	`df_pivot = df.pivot_table(values='Value', index='Category', aggfunc='sum')`
80	Visualization	Basic Plotting	*.plot()*	kind: Type of plot (e.g., 'line', 'bar', 'hist') x: Column to use for x-axis (optional) y: Column to use for y-axis (optional) title: Title of the plot.	The `.plot()` function is used to create a basic plot (line, bar, histogram, etc.) from a DataFrame.	`df['Value'].plot(kind='line')`
81	Visualization	Histogram	*.hist()*	bins: Number of bins for the histogram rwidth: Width of bars in the histogram.	The `.hist()` function is used to create a histogram for a column.	`df['Value'].hist(bins=10)`
82	Visualization	Box Plot	*.boxplot()*	column: Column(s) to plot by: Column to group by for the box plot.	The `.boxplot()` function is used to create a box plot.	`df.boxplot(column='Value', by='Category')`
83	Visualization	Scatter Plot	*.scatter()*	x: Column for x-axis y: Column for y-axis c: Color of the points (optional) s: Size of the points (optional).	The `.scatter()` function is used to create a scatter plot.	`df.plot.scatter(x='Category', y='Value')`
84	Visualization	Pie Chart	*.pie()*	autopct: String to format the percentage display startangle: Angle to start the pie chart (optional).	The `.pie()` function is used to create a pie chart.	`df['Category'].value_counts().plot.pie(autopct='%1.1f%%')`
85	Visualization	Hexbin Plot	*.hexbin()*	gridsize: Number of hexagons in the plot cmap: Colormap to use for coloring the plot.	The `.hexbin()` function is used to create a hexagonal bin plot.	`df.plot.hexbin(x='Category', y='Value', gridsize=20)`