Python Data Processing Commands

|  |  |  |  |
| --- | --- | --- | --- |
| **Method/Command** | **Source Library** | **Example** | **Description** |
| open() | Inbuilt | open("sample.csv", 'r’) | Open file in read mode and return stream |
| readline() | Inbuilt | infile.readline() | Returns line from file |
| detect() | chardet | chardet.detect(f.read(100000)) | Find encoding and other details by reading 100KB of file data |
| read\_csv() | pandas | pandas.read\_csv(  "sample.csv",  encoding=”ascii”,  sep=",",  on\_bad\_lines="warn",  engine="python"  ) | Read csv file and return data frame |
| head() | pandas | data\_frame.head() | Return top 5 rows of data frame |
| <data frame variable>[<column name>] | pandas | data\_frame['SepalLength'] | Return mentioned column values from data frame |
| to\_numeric() | Pandas | df[col] = pandas.to\_numeric(df[col], errors="coerce") | Convert input to numeric type |
| <data frame variable>[<column name>].median() | Pandas | df["SepalWidth"].median() | Returns median value from the given column’s values |
| fillna() | Pandas | df["SepalWidth"] = df["SepalWidth"].fillna(df["SepalWidth"].median()) | Fill null values of the mentioned column with median value |
| describe() | Pandas | df.describe() | Generate descriptive statistics like min, max, count, mean and others |
| rename() | Pandas | df = df.rename(columns={  "SepalWidth": "SepalWidth2",  "SepalLength": "SepalLength2"  }) | Change column names of data frame |
| writer() | csv | writer = csv.writer(f\_out) | Takes file reference and creates writer object |
| writerow() | csv | writer.writerow(headers) | Adds given variable data to the file through writer object |
| read\_excel() | Pandas | df = pandas.read\_excel("./dataset\_samples/Customer\_Info.xlsx") | Read excel file |
| between() | Pandas | df["AgeValid"] = df["Age"].between(18, 99) | For each row validates age and returns Boolean value |
| contains() | Pandas | df["EmailValid"] = df["Email"].str.contains(r"^[^@]+@[^@]+\.[^@]+$", na=False) | Considers each value of the column as string and checks for mentioned expression. Returns Boolean for each value |
| r"^[^@]+@[^@]+\.[^@]+$" | Inbuilt | df["EmailValid"] = df["Email"].str.contains(r"^[^@]+@[^@]+\.[^@]+$", na=False) | **^** to start of the string  **[^@]+** Match one or more characters that are NOT '@'  **@** Literal '@' character  **[^@]+** Again, match one or more characters that are NOT '@'  **\.** Literal dot (.), escaped so it’s not treated as “any character”  **[^@]+** Match one or more non-@ characters (like “com”)  **$** Anchors the pattern to end of the string |
| astype() | Pandas | df["PurchaseAmount"] = (  df["PurchaseAmount"]  .astype(str)  .str.replace(",", "", regex=False)  .replace("N/A", pd.NA)  .astype(float)  ) | Convert everything to string, float |
| replace() | Pandas | df["PurchaseAmount"] = (  df["PurchaseAmount"]  .astype(str)  .str.replace(",", "", regex=False)  .replace("N/A", pd.NA)  .astype(float)  ) | Replace a string with new string |
| strip() | Pandas | df["Name"] = df["Name"].str.strip() | Trim whitespace from names |
| to\_datetime | Pandas | df["JoinDate"] = pd.to\_datetime(df["JoinDate"], errors="coerce") | Convert JoinDate to date and time |
| timestamp() | Pandas | pandas.timestamp("2022-01-01") | Convert string to date and time |
| to\_excel() | Pandas | df.to\_excel("./dataset\_samples/Customer\_Info\_Cleaned.xlsx", index=False) | Create excel from data frame |
| to\_csv | Pandas | df.to\_csv("./dataset\_samples/Customer\_Info\_Cleaned.csv", index=False, encoding="utf-8") | Create csv from data frame |
| load() | Json | with open("./dataset\_samples/D3\_Customer\_Orders.json") as f:  data = json.load(f) | Creates a list with dictionaries |
| json\_normalize | Pandas | df = pd.json\_normalize(  data,  record\_path=["orders"],  meta=[  ["customer", "id"],  ["customer", "name"],  ["customer", "email"],  ["customer", "address", "city"],  ["customer", "address", "zip"]  ],  meta\_prefix="",  errors="ignore"  ) | Flatten the Orders with Customer Info. This gives you one row per order, with customer details included in each row. |
| columns.str.replace() | Pandas | df.columns = df.columns.str.replace(r"\.", "\_", regex=True) | Replace text in column names |
| Multiplication | Pandas | df["total\_value"] = df["price"] \* df["quantity"] | Multiply column values and create new column |
| groupby() with count() | Pandas | df.groupby("customer\_name")["item"].count() | Count of items ordered by each customer |
| groupby() with sum() | Pandas | df.groupby("customer\_id")["total\_value"].sum() | Total spend per customer |
| Filter Operation | Pandas | df[df["delivery\_status"] == "Delivered"] | Filter only delivered orders |
| Filter Operation with Date | Pandas | df[pd.to\_datetime(df["delivery\_date"], errors="coerce") > "2023-06-03"] | Orders after June 3rd |
| to\_excel() with multiple sheets | Pandas | df\_delivered = df[df["delivery\_status"] == "Delivered"]  df\_pending = df[df["delivery\_status"] == "Pending"]  with pd.ExcelWriter("./dataset\_samples/D3\_Flattened\_Orders\_Split.xlsx") as writer:  df\_delivered.to\_excel(writer, sheet\_name="Delivered", index=False)  df\_pending.to\_excel(writer, sheet\_name="Pending", index=False) | Export to multiple sheets in same excel |
| len() | Inbuilt | summary = {  "Total Orders": len(df),  "Unique Customers": df["customer\_id"].nunique(),  "Total Spend": df["total\_value"].sum()  } | Get number of records of data frame |
| nunique() | Pandas | df["customer\_id"].nunique() | Unique Customers count |
| sum() | Pandas | df["total\_value"].sum() | Sum of the given column values |
| sort\_values() | Pandas | df\_sorted= df.sort\_values(ascending=False, by="TotalSales").head(3) **OR** df\_sorted=df.sort\_values("TotalSales", ascending=False).head(3) **OR** df.sort\_values("TotalSales", ascending=False, inplace=True) | Sort values in data frame. If memory is less, inplace helps to modify the existing df |
| read\_excel() | Pandas | df = pd.read\_excel("monthly\_sales.xlsx", sheet\_name="Sheet2") | Read only specific sheet |
| read\_excel() with parsing dates | Pandas | df = pd.read\_excel(  "employee\_records.xlsx",  parse\_dates=["JoiningDate", "LastLogin"]  ) | Read excel file along by parsing dates fields |
| read\_excel() all sheets | Pandas | dfs = pd.read\_excel("file.xlsx", sheet\_name=None) | Read all sheets of excel. It returns dictionary of sheets. Each key is sheet name and value is data frame |
| query() | Pandas | df\_filtered = df.query(  "LastLogin > '2023-07-01' and Status in ['Remote', 'Present']"  ) | Filter rows |
| parse() | xml.etree.ElementTree | tree = ET.parse("./dataset\_samples/D4\_Employees.xml") | Load and parse XML |
| getroot() | xml.etree.ElementTree | tree.getroot() | Get root of xml |
| tag | xml.etree.ElementTree | root.tag | Gives root tag value |
| findall() | xml.etree.ElementTree | for emp in root.findall("employee") | Get all values under employee |
| get() | xml.etree.ElementTree | emp.get("id") | Get attribute value of node |
| find() | xml.etree.ElementTree | name\_tag = emp.find("name") | Get sub-elements, get name |
| text | xml.etree.ElementTree | name\_tag.text | Get value of the tag |
| attrib | xml.etree.ElementTree | "title" in name\_tag.attrib | Gives dictionary of element’s attributes |
| findall() | xml.etree.ElementTree | root.findall(".//employee[@id='E002']") | XPath-style filtering — find specific employee by ID |
| ET.fromstring() | xml.etree.ElementTree | tree\_from\_string = ET.ElementTree(ET.fromstring(xml\_string)) | Tree from string |
| append() | xml.etree.ElementTree | root.append(new\_emp) | Add new element to root |
| write() | xml.etree.ElementTree | tree.write("./dataset\_samples/D4\_Employees.xml", encoding="utf-8", xml\_declaration=True) | Update file/if file not present create new |
| ET.Element() | xml.etree.ElementTree | new\_emp = ET.Element("employee", id="E004") | Create new element |
| ET.SubElement() | xml.etree.ElementTree | ET.SubElement(new\_emp, "name", title="Lead").text = "Kiran" | Add sub-element |
| dataframe() | Pandas | df = pandas.DataFrame({  'id': [1, 2, 3],  'name': ['Sandeep', 'Aditi', 'Rohan']  }) | DataFrame creation |
| to\_parquet() | Pandas | df.to\_parquet('./dataset\_samples/D5\_sample.parquet', index=False) | Write DataFrame to a Parquet file (columnar format, efficient for analytics) |
| read\_parquet() | Pandas | df\_parquet = pd.read\_parquet('./dataset\_samples/D5\_sample.parquet') | Read the Parquet file into a DataFrame |
| to\_csv with compression | Pandas | df.to\_csv('./dataset\_samples/D5\_compressed\_data.csv.gz', index=False, compression='gzip') | Write DataFrame to a GZIP-compressed CSV file |
| read\_csv with compression | Pandas | df\_gzip = pd.read\_csv('./dataset\_samples/D5\_compressed\_data.csv.gz', compression='gzip') | Read the GZIP-compressed CSV file |
| dict() | Inbuilt | compression\_opts = dict(method='zip', archive\_name='nested.csv') | Define zip compression options: specify filename inside the archive |
| to\_csv with compression options | Pandas | df.to\_csv('./dataset\_samples/D5\_archive.zip', index=False, compression=compression\_opts) | Write DataFrame into a ZIP archive containing 'nested.csv' |
| read\_csv of zip file | Pandas | df\_zip = pd.read\_csv('./dataset\_samples/D5\_archive.zip') | Read CSV file from ZIP archive using zip:// protocol. Zip:// is used for multiple files, if single file its not required |
| ZipFile(), write() | Zipfile | with zipfile.ZipFile('./dataset\_samples/D5\_multi\_archive.zip', 'w') as zipf:  zipf.write('./dataset\_samples/D5\_compressed\_data.csv.gz', arcname='compressed.csv.gz')  zipf.write('./dataset\_samples/D5\_sample.parquet', arcname='data.parquet')  zipf.write('./dataset\_samples/D5\_archive.zip', arcname='original\_archive.zip') | Create a zip archive with multiple files |
| guess\_type() | mimetypes | file\_path = './dataset\_samples/D5\_compressed\_data.csv.gz'  print(mimetypes.guess\_type(file\_path)) | Detect file compression format |
| ParquetFile() | pyarrow.parquet | metadata = pq.ParquetFile('./dataset\_samples/D5\_sample.parquet').metadata  print(metadata.schema) | Gives schema, like type of fields it contains |
| Concat() | Pandas | df\_combined = pd.concat([df\_csv, df\_json, df\_par, df\_xl, df\_xml], ignore\_index=True) | Combine data frames |
| Drop\_duplicates() | Pandas | df\_combined.drop\_duplicates(inplace=True) | Drop duplicates and overwrite the same data frame using inplace |
| Shape | Pandas | df.shape # returns (rows, columns)  before = df\_combined.shape[0] | It’s a property that returns a tuple with number of rows and columns in a data frame |
| Apply() | Pandas | df\_combined["valid\_email"] = df\_combined["email"].apply(validate\_email\_format) | apply a function element-wise or row-wise/column-wise depending on your target. In the example calls the validate\_email\_format method and passes email value as parameter |
| Reset\_index() | Pandas | df\_combined.reset\_index(drop=True, inplace=True) | Reset df indexes |
| Index | Pandas | df\_combined["id"] = df\_combined.index + 1 | Property of data frame |
| df.isnull() | Pandas |  | Returns a DataFrame of True/False for NaNs |
| df.isnull().sum() | Pandas |  | Count of missing values per column |
| df.notnull() | Pandas |  | Opposite of isnull() |
| df.dropna() | Pandas |  | Drops rows with any NaN |
| df.dropna(axis=1) | Pandas |  | Drops columns with any NaN |
| df.dropna(how='all') | Pandas |  | Drops rows where all values are NaN |
| df.dropna(thresh=2) | Pandas |  | Keeps rows with at least 2 non-NaN values |
| df.dropna(subset=['col1']) | Pandas |  | Drops rows where 'col1' is NaN |
| df.fillna(0) | Pandas |  | Replace NaNs with 0 |
| df.fillna(method='ffill') | Pandas |  | Forward fill |
| df.fillna(method='bfill') | Pandas |  | Backward fill |
| df['col'].fillna(df['col'].mean()) | Pandas |  | Fill with column mean |
| df.duplicated() | Pandas |  | Marks duplicate rows as True |
| df.duplicated(subset=['col']) | Pandas |  | Checks duplicates based on specific column(s) |
| df.drop\_duplicates(subset=['col']) | Pandas |  | Removes based on specific column(s) |
| df.drop\_duplicates(keep='last') | Pandas |  | Keeps the last occurrence |
| re.sub() | re | text = re.sub(r'\s+', ' ', text).strip() | Substitute the pattern with given text |
| re.findall() | re | re.findall(r'#\w+', raw\_updated) | Find all matching pattern and return in list |
| nltk.download() | nltk | nltk.download('stopwords') | Download stopwords package from natural language toolkit |
| stopwords.words() | nltk | stopwords.words('english') | Get English stop words like a, the |
| re.compile() | Re | pattern = re.compile(r'''  (?P<url> # --- Match URLs ---  https?://\S+  )  |  (?P<special> # --- Match Unwanted Special Characters ---  [^  \w # Word characters  \s # Whitespace  \#\.@ # Keep #, . and @  ]+  )  ''', re.VERBOSE) | Compile a regular expression pattern and return pattern object |
| Strptime() | datetime | datetime.strptime(date\_str, fmt) | Returns datetime value from string using the format |
| to\_datetime() | Pandas | df['standardized\_date'] = pd.to\_datetime(df['raw\_date'], errors='coerce', dayfirst=True) | Converts datetime string value to datetime value |
| Timezone() | pytz | local\_tz = pytz.timezone('Asia/Kolkata') | Return time zone info |
| UTC | pytz | utc\_tz = pytz.UTC | Set timezone variable |
| Localize() | pytz | localized\_dt = local\_tz.localize(dt) | Get localized datetime value |
| Astimezone() | datetime | utc\_dt = localized\_dt.astimezone(utc\_tz) | Convert a datetime to mentioned zone datetime value |
| Replace() | datetime | dt = datetime(2025, 7, 12, 9, 20)  dt\_local = dt.replace(tzinfo=ZoneInfo('Asia/Kolkata')) | Replace the datetime values with given values |
| tz\_convert() | Inbuilt | dt\_utc = pd.to\_datetime(timestamp\_str, utc=True)  dt\_local = dt\_utc.tz\_convert(ZoneInfo("Asia/Kolkata")) | Convert datetime value to different timezone |
| Re-order columns | Pandas | desired\_order = ['customer\_id', 'order\_date', 'amount']  df = df[desired\_order] | Re-order columns |
| Info() | Pandas | df.info() | Give data frame info like type, range of indexes, columns and their types and memory usage |
| convert\_dtypes() | Pandas | df = df.convert\_dtypes() | Convert data types of data based on data values |
| Quantile() | Pandas | Q1 = df[col].quantile(0.25) | It returns the value below which 25% of the data falls. |
| Zscore() | Scipy.stats | z\_scores = stats.zscore(df[col]) | Compute z score for each value in the sample, relative to sample mean and standard deviation.  **Behind-the-Scenes** Steps:  - Calculate the mean (μ) of the column  - Calculate the standard deviation (σ)  - For each value xᵢ, compute:  zᵢ = \frac{xᵢ - μ}{σ} |
| Copy() | Pandas | df\_copy = df.copy() | Creates copy of Data frame |
| df.loc[<row\_selector>, <column\_selector>] | Pandas | df\_copy.loc[(df[col] < lower) | (df[col] > upper), col] = median | - .loc[...] → selects those rows within the specified column  - = median → replaces all identified outlier values with the precomputed median. Locates the rows where the condition is met and writes the median value into that column |
| df[col].clip(lower, upper) | Pandas | df\_copy[col] = df[col].clip(lower, upper) | - lower: minimum allowed value  - upper: maximum allowed value  - Any value below lower is replaced with lower  - Any value above upper is replaced with upper  - Values within bounds are untouched |
| Iterrows() | Pandas | for i, row in df.iterrows() | Loop through rows |
| Isinstance() | Inbuilt | if not isinstance(row['age'], int): | Check date type |
| Seed() | Numpy | np.random.seed(42) | Tries to keep same random values through seeding |
| Normal() | Numpy | np.random.normal(loc=50000, scale=15000, size=100) | Normally distributed 100 salaries |
| Randint() | Numpy | np.random.randint(1, 30, size=100) | Random 100 years of experience |
| Choice() | Numpy | np.random.choice(['HR', 'Finance', 'Tech', 'Sales'], size=100) | Random department values |
| Round() | Pandas | df['salary'] = df['salary'].round(2) or df.describe().round(2) | Round off all salary column values. Round off all values from describe |
| Describe() | Pandas | df.describe(include='all') | Include non-numeric columns as well in stats |
| Histplot() | seaborn | sns.histplot(df['salary'], kde=True) | Draw histogram chart along with smooth KDE curve |
| Title() | matplotlib.pyplot | plt.title("Salary Distribution")  plt.xlabel("Salary")  plt.ylabel("Frequency")  plt.show() | Plot chart with given values and show it |
| Skew() | Pandas | df['salary'].skew() | Measures how symmetrical your data is around the mean.  Zero skew → perfectly symmetrical (e.g., normal distribution).  Positive skew → tail extends to the right (many low values, few high ones).  Negative skew → tail extends to the left (many high values, few low ones). |
| Kurt() | Pandas | df['salary'].kurt() | Definition: Measures the sharpness of the peak and the heaviness of the tails.  Low kurtosis (<3) → flat distribution (platykurtic), fewer outliers.  High kurtosis (>3) → sharper peak (leptokurtic), more values in tails → more outliers. |
| Quantile() | Pandas | df['salary'].quantile([0.25, 0.5, 0.75]) | Computes the 25th, 50th, and 75th percentiles of the salary column, which are also known as:  - 0.25 → First Quartile (Q1): 25% of salaries fall below this value  - 0.50 → Median (Q2): Middle value—half the salaries are below, half are above  - 0.75 → Third Quartile (Q3): 75% of salaries fall below this value |
| Qcut() | Pandas | df['salary\_group'] = pd.qcut(df['salary'], q=4, labels=['Low', 'Mid-Low', 'Mid-High', 'High']) | Binning salaries into quartile groups. Stands for quantile cut—it splits a numeric column (like salary) into equal-sized groups based on their distribution. |
| Value\_counts() | Pandas | df['salary\_group'].value\_counts() | Quick check of how many records fall into each bin |
| Groupby() | Pandas | total\_sales\_by\_region = df.groupby('Region')['Sales'].sum() | Group by region and sum of sales |
| Pivot\_table() | Pandas | pivot = df.pivot\_table(values='Sales', index='Region', columns='Month', aggfunc='sum', fill\_value=0) | Pivot table with region wise monthly sales sum |
| Groupby multiple columns | Pandas | summary = df.groupby(['Region', 'Product'])['Sales'].agg(['sum', 'mean', 'count']) | Group by region and product and give sum of sales, mean & count |
| Barplot() | Seaborn | sns.barplot(x=total\_sales\_by\_region.index, y=total\_sales\_by\_region.values) | Bar plot with given values |
| Melt() | Pandas | pivot\_reset = pivot.reset\_index().melt(id\_vars='Region', var\_name='Month', value\_name='Sales') | De-normalize rows |
| Barplot() | Seaborn | sns.barplot(data=pivot\_reset, x='Region', y='Sales', hue='Month') | Monthly Sales by Region |
| Melt() | Pandas | summary\_melted = summary.reset\_index().melt(  id\_vars=['Region', 'Product'],  value\_vars=['sum', 'mean', 'count'],  var\_name='Metric',  value\_name='Sales'  ) | De-normalize rows |
| Create new column and data frame | Pandas | summary\_melted['Group'] = summary\_melted['Region'] + ' - ' + summary\_melted['Product']  sub\_summary=summary\_melted[['Group', 'Metric', 'Sales']] | Create new column and create new data frame with few columns |
| Barplot() | Seaborn | sns.barplot(  data=sub\_summary,  x='Group', y='Sales',  hue='Metric'  ) | Sales Summary by Region and Product (sum, mean, count) |
| Merge() | Pandas | inner\_join = pd.merge(df\_customers, df\_orders, on='CustomerID', how='inner') | SQL-style inner join |
| Merge() | Pandas | left\_join = pd.merge(df\_customers, df\_orders, on='CustomerID', how='left') | Left join |
| Merge() | Pandas | outer\_join = pd.merge(df\_customers, df\_orders, on='CustomerID', how='outer') | Outer join |
| Merge() | Pandas | df\_left = pd.DataFrame({'ID1': [1, 2], 'Value1': [10, 20]})  df\_right = pd.DataFrame({'ID2': [2, 3], 'Value2': [30, 40]})  custom\_merge = pd.merge(df\_left, df\_right, left\_on='ID1', right\_on='ID2', how='outer') | Multi joins |
| Map() | Pandas | codebook = {'A': 'Admin', 'B': 'Business', 'C': 'Customer'}  df['UserTypeLabel'] = df['UserType'].map(codebook).fillna('Unknown') | Map values and create new column |
| Unstuck() | Pandas | grouped = df.groupby('Region')['UserTypeLabel'].value\_counts().unstack().fillna(0).astype(int) | Group by region with columns user type label and give count under each label |
| Date\_range() | Pandas | date\_range = pd.date\_range(start='2023-01-01', end='2023-03-31', freq='D') | Create data frame with with given date ranges and frequency |
| Randn() | Numpy | temps = (25 + np.random.randn(len(date\_range)) \* 5).round(2) | Create a list of values with randn which take number of values to be generated. |
| Resample() | Pandas | df['weekly\_avg'] = df['temperature'].resample('W').mean().round(2)  df['monthly\_avg'] = df['temperature'].resample('ME').mean().round(2) | Resample - weekly average and monthly average. Generates mean for every week/month |
| Rolling() | Pandas | df['rolling\_7d'] = df['temperature'].rolling(window=7).mean().round(2)  df['rolling\_14d'] = df['temperature'].rolling(window=14).mean().round(2) | Rolling averages - 7-day and 14-day windows. Adds 7/14 values till that day. Output Breakdown   | **Day** | **Sales** | **Rolling Avg (3-day mean)** | | --- | --- | --- | | Jan 1 | 10 | NaN | | Jan 2 | 20 | NaN | | Jan 3 | 30 | (10+20+30)/3 = **20.0** | | Jan 4 | 40 | (20+30+40)/3 = **30.0** | | Jan 5 | 50 | (30+40+50)/3 = **40.0** | | Jan 6 | 60 | (40+50+60)/3 = **50.0** | | Jan 7 | 70 | (50+60+70)/3 = **60.0** | |
| Contains() | Pandas | keywords = ['irrigation', 'technology']  pattern = '|'.join(keywords)  df[df['category'].str.contains(pattern, case=False, na=False)] | Search with multiple values |
| Date\_range() | Pandas | dates = pd.date\_range('2023-01-01', periods=100, freq='D') | Get 100 dates starting from given point with daily frequency |
| Figure() | matplotlib.pyplot | plt.figure(figsize=(10, 4)) | Set plot size in inches |
| Plot() | matplotlib.pyplot | plt.plot(df['date'], df['sales'], color='teal', label='Daily Sales') | Plot the diagram. Gives line graph |
| Histplot() | Seaborn | sns.histplot(df['temperature'], bins=20, color='coral', kde=True) | Generate bar graph with 20 bars |
| Heatmap() | Seaborn | corr = df.drop(columns='date').corr()  sns.heatmap(corr, annot=True, cmap='coolwarm', fmt='.2f') | Remove non-numeric values and plot heatmap |
| Astype() | Pandas | df['sex'] = df['sex'].astype('category') | Category is also a data type. Categorical type (optimizes memory and performance). Behind the scenes, pandas stores category labels like 'male' and 'female' as integer codes linked to a lookup table. It’s like giving your data compression + smart indexing. |
| df.isnull().sum() | Pandas | print("\nMissing Values:\n", df.isnull().sum()) | It counts number of nulls in each column. Df.isnull creates df with Boolean values, if null then true else false. Sum will add all those true values. True = 1 here. So it’s like it counts all null values using sum |
| Boxplot() | Seaborn | sns.boxplot(x='class', y='fare', data=df) | Box plot |
| Countplot() | Seaborn | sns.countplot(x='sex', data=df) | Count plot |
| Violinplot() | Seaborn | sns.violinplot(x='class', y='age', data=df) | Violin plot |
| Catplot() | Seaborn | sns.catplot(x='embarked', hue='survived', data=df, kind='count') | Categorical Exploration |
| Cut() | Pandas | df['age\_bin'] = pd.cut(df['age'], bins=[0, 12, 18, 40, 60, 80]) | Creates age ranges column for each row. Age bin column will have values (0,12) or (12,18) or (18,40) or (40,60) or (60,80) |
| Pairplot() | Seaborn | sns.pairplot(df[['age', 'fare', 'survived']], hue='survived') | Pair plot |
| Idxmax() | Pandas | top\_city = df.groupby('store\_city')['total\_amount'].sum().idxmax() | Top performing city |
| Rolling() | Pandas | monthly\_rev['rolling\_avg'] = (  monthly\_rev  .groupby('store\_city')['total\_amount']  .rolling(window=2, min\_periods=1)  .mean()  .round(2)  .reset\_index(level=0, drop=True)  ) | Get rolling for every two months. Even if 1 month data is present |
| Isin() | Pandas | df['is\_loyal\_segment'] = df['customer\_segment'].isin(['Gold', 'Silver']) | Check in multiple values |
| Day\_name() | Pandas | df['txn\_day'] = df['txn\_date'].dt.day\_name() | Get day name for date |
| Weekday | Pandas | df['is\_weekend'] = df['txn\_date'].dt.weekday >= 5 | Get weekday number |
| Get\_dummies() | Pandas | pd.get\_dummies(df['store\_city'], prefix='city') | Create columns for each city and mark rows with that particular city as true |
| Get\_dummies() | Pandas | df\_encoded = pd.get\_dummies(df, columns=['store\_city'], prefix='city') | Create columns for each city and mark rows with that particular city as true |
| fit\_transform() | sklearn.preprocessing.LabelEncoder | le = LabelEncoder()  df['segment\_encoded'] = le.fit\_transform(df['customer\_segment']) | Label encoder, give a number to each value |
| Safe\_load() | yaml | config = yaml.safe\_load(file) | Load yaml file |
| Is\_integer\_dtype() | Pandas | if not pd.api.types.is\_integer\_dtype(df['customer\_id']):  raise TypeError("customer\_id must be integer") | Check column data type |
| Is\_numeric\_dtype() | Pandas | if not pd.api.types.is\_numeric\_dtype(df['total\_amount']):  raise TypeError("total\_amount must be numeric") | Check column data type |
| Any() | Pandas | if (df['total\_amount'] <= 0).any():  raise ValueError("Some amounts are non-positive") | If anyone is negative |
| Connect() | sqlite3 | conn = sqlite3.connect(db\_path) | Create connection |
| To\_sql() | Pandas | conn = sqlite3.connect(db\_path)  df.to\_sql(table\_name, conn, if\_exists='replace', index=False) | Load data frame records into table |
| from\_service\_account\_file() | from google.oauth2 import service\_account | #scopes = ['https://www.googleapis.com/auth/devstorage.read\_write']  scopes = ['https://www.googleapis.com/auth/cloud-platform'] creds = service\_account.Credentials.from\_service\_account\_file(  filename="C:/Users/kodur/source/repos/PyDataEngi/key.json",  scopes=scopes  ) | Get credentials of gcp |
| GCSFileSystem() | gcsfs | fs = gcsfs.GCSFileSystem(token=creds) | Creates a Python object called fs, which acts as a virtual filesystem interface to Google Cloud Storage (GCS). |
| Ls() | gcfs | print(fs.ls("sandeep-data-bucket")) | Lists files/folders in given bucket |
| Open() | gcfs | timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  path\_gcs = f"sandeep-data-bucket/converted/D16\_sales\_data\_v{timestamp}.parquet"  with fs.open(path\_gcs, 'wb') as f:  df.to\_parquet(f, engine="pyarrow", index=False) | Upload file to GCP |
| Exists() | gcfs | exists = fs.exists(full\_path) | Valid file existence in path |
| To\_gbq() | from pandas\_gbq import to\_gbq | PROJECT\_ID = "tranquil-post-461304-m7"  DATASET = "sandeep\_dataset"  to\_gbq(product\_df, f"{DATASET}.product\_data", project\_id=PROJECT\_ID, if\_exists="replace")  to\_gbq(pricing\_df, f"{DATASET}.pricing\_data", project\_id=PROJECT\_ID, if\_exists="replace") | Creates two tables |
| Client() | from google.cloud import bigquery | client = bigquery.Client() | Big query client instance |
| Query() | from google.cloud import bigquery | joined\_df = client.query(query).to\_dataframe() | Query returns a query job and executes it. Output is converted to data frame |
| Table(), create\_table() | from google.cloud import bigquery | table\_id = f"{PROJECT\_ID}.{DATASET}.product\_data\_partitioned"  schema = [  bigquery.SchemaField("product\_id", "INTEGER"),  bigquery.SchemaField("category", "STRING"),  bigquery.SchemaField("price", "FLOAT"),  bigquery.SchemaField("created\_at", "DATE"),  ]  table = bigquery.Table(table\_id, schema=schema)  client.create\_table(table, exists\_ok=True) | Create a big query table variable with given table name and schema |
| Timepartitioning() | from google.cloud import bigquery | table.time\_partitioning = bigquery.TimePartitioning(field="created\_at") | Create partitions |
| Cluster\_fields | bigquery | table.clustering\_fields = ["category"] | Create clusters. When used for partitioned table, clusters will be created inside partitions |
| Create\_table() | bigquery | table = client.create\_table(table, exists\_ok=True) | Create table with given details inside GCP BigQuery |
| Load\_data\_from\_dataframe() | bigquery | client.load\_table\_from\_dataframe(product\_df, table).result() | Load data to table from data frame using job and result will execute the job |
| create\_dataset() | bigquery | client = bigquery.Client()  project\_id = "tranquil-post-461304-m7"  dataset\_id = "etl\_logs"  full\_dataset\_id = f"{project\_id}.{dataset\_id}"  # Step 1: Create Dataset if Needed  dataset = bigquery.Dataset(full\_dataset\_id)  dataset.location = "asia-south1" # or your preferred region  client.create\_dataset(dataset, exists\_ok=True) | Create database dataset [schema] |
| Topic\_path(), create\_topic() | from google.cloud import pubsub\_v1 | topic\_id = "etl-alerts"  publisher = pubsub\_v1.PublisherClient()  topic\_path = publisher.topic\_path(project\_id, topic\_id)  topic = publisher.create\_topic(request={"name": topic\_path}) | Create topic |
| SubscriberClient(),subscription\_path(),subscribe(), | from google.cloud import pubsub\_v1 | PROJECT\_ID = "tranquil-post-461304-m7"  SUBSCRIPTION\_ID = "errorsubs"  subscriber = pubsub\_v1.SubscriberClient()  subscription\_path = subscriber.subscription\_path(PROJECT\_ID, SUBSCRIPTION\_ID) | Listen to subscribed messages |
| Result() | pubsub\_v1 | streaming\_pull\_future.result(timeout=60) | Listen for 60 seconds |
| StreamHandler(),JsonFormatter(), | logging | logger = logging.getLogger("structuredLogger")  logHandler = logging.StreamHandler()  formatter = jsonlogger.JsonFormatter('%(asctime)s %(levelname)s %(message)s %(pipeline\_step)s')  logHandler.setFormatter(formatter)  logger.addHandler(logHandler)  logger.setLevel(logging.INFO) | Set up structured logging in json format |
| blob.download\_as\_text(),json.dumps(record), blob.upload\_from\_string() | from google.cloud import storage, json | blob = storage\_client.bucket(BUCKET\_NAME).blob(BLOB\_NAME)  existing = blob.download\_as\_text() if blob.exists() else ""  new\_content = existing + json.dumps(record) + "\n"  blob.upload\_from\_string(new\_content, content\_type="application/json") | Download json file from storage bucket, append new record and upload back as json format |
| Insert\_rows\_json() | bigquery | bigquery\_client.insert\_rows\_json(BQ\_TABLE, [record]) | Insert data into big query table |
| Publish() | Json, pubsub\_v1 | data = json.dumps({"alert": message}).encode("utf-8")  publisher.publish(TOPIC\_PATH, data=data) | Publish the message to topic |
| Info() | logging | logger.info("Processing row", extra={"pipeline\_step": "read\_row"}) | Send log info to logger |
| Isna() | Pandas | if pd.isna(row["product\_id"]):  raise ValueError("Missing product\_id") | Checks if column is null |
| To\_json() | Pandas | row.to\_json() | Converts row of data frame to json |
| Isoformat() | Datetime | datetime.utcnow().isoformat() | Current time in YYYY-MM-DD |
| Error() | logging | logger.error("Failure encountered", extra={"pipeline\_step": "error\_handler"}) | Log the errors |
| getLogger() | logging | for handler in logging.getLogger().handlers:  if hasattr(handler, "flush"):  handler.flush() | Ensure all log info is flushed |
|  |  |  |  |