Data/Parameters You Might Need

Here are categories of data you'd want to collect and later analyze:

1. Device Metadata
    - Device ID, type (router, switch, firewall, etc.)
    - Manufacturer, model, serial number
    - Installation date / age of device
    - Software/firmware version
2. Usage & Performance Metrics
    - CPU utilization over time
    - Memory utilization
    - Disk/storage utilization (if applicable)
    - Network throughput (packets/sec, bandwidth usage)
    - Error rates (dropped packets, retransmissions, collisions)
    - Latency and jitter
3. Environmental & Operational Data
    - Device location (datacenter, branch office, region)
    - Power supply metrics (voltage fluctuations, outages)
    - Temperature / overheating events
    - Cooling system status (fans, airflow sensors)
    - Uptime since last reboot
4. Health & Status Indicators

- System logs (error/warning messages)
- SNMP traps or alerts
- Number of reboots / crashes
- Firmware/software bugs reported
- Security incidents (failed logins, intrusion attempts)

5. Failure & Maintenance History
    - Date/time of past failures
    - MTBF (Mean Time Between Failures)
    - MTTR (Mean Time To Repair)
    - Maintenance activities (patching, firmware updates, component replacement)
    - Warranty/AMC coverage
6. External Factors
    - Workload spikes (seasonal traffic increase, sudden surges)
    - Network topology (is the device in a critical bottleneck?)
    - Dependency on other devices

Data Identification:

For a sensor, `disk_usage_%` might always be `NaN`.

For a router, `sensor_faults_detected` might always be `0` or `N/A`.

Pandas Cheatsheet:

import pandas as pd

1) Read a csv file: pd.read_csv("Pandas/NetflixDataset.csv")
2) Get top records: .head(count), .tail(count)

3]:
```
df = pd.read_csv("NetflixDataset.csv").tail(2)
df
```

3]:

	Show_Id	Category	Title	Director	Cast	Country	Release_Date	Rating	Duration	Type	Description
7787	s7786	TV Show	Zumbo's Just Desserts	NaN	Adriano Zumbo, Rachel Khoo	Australia	October 31, 2020	TV-PG	1 Season	International TV Shows, Reality TV	Dessert wizard Adriano Zumbo looks for the nex...
7788	s7787	Movie	ZZ TOP: THAT LITTLE OL' BAND FROM TEXAS	Sam Dunn	NaN	United Kingdom, Canada, United States	March 1, 2020	TV-MA	90 min	Documentaries, Music & Musicals	This documentary delves into the mystique behi...

3) df = pd.read_csv("NetflixDataset.csv").tail(2)
The above command returns a dataframe: It has **rows and columns** (like a table), and you can use it to **store, analyze, and process data** easily.

#To create dataframe in python:

```
# Dictionary of data
data = {
    "Movie": ["Inception", "Titanic", "Avatar"],
    "Year": [2010, 1997, 2009],
    "Rating": [8.8, 7.9, 7.8]
}

# Convert dictionary into DataFrame
df = pd.DataFrame(data)
```

Convert list to dataframe:
df = pd.DataFrame(list, cols)

0) Setup & quick tips

```
import pandas as pd
import os

print(os.getcwd())     # where your notebook is running
print(os.listdir())    # what files are here
pd.set_option("display.max_rows", 8)
pd.set_option("display.max_columns", 20)
```

Paths:

- "data.csv" → file in the current folder

- "folder/data.csv" → relative path
- "/absolute/path/data.csv" → absolute path (starts with / on Linux/Mac, with a drive like C:\ on Windows)

---

1) EXTRACT (read/import data)

CSV (most common)

```
df = pd.read_csv(
    "sales.csv",
    usecols=["order_id", "date", "country", "amount"],  # read only needed cols
    dtype={"order_id": "Int64", "country": "string"},    # initial dtypes
    parse_dates=["date"],                                # parse to datetime
    na_values=["", "NA", "null", "?"],                   # treat these as NaN
    thousands=",",                                       # "1,234" -> 1234
    encoding="utf-8",                                    # or "latin-1"
)
```

Large files (stream in chunks)

```
chunks = pd.read_csv("huge.csv", chunksize=100_000)
df = pd.concat(
    (c[c["amount"] > 0] for c in chunks),    # example pre-filter
    ignore_index=True
)
```

Excel

```
df = pd.read_excel("book.xlsx", sheet_name="Sheet1", engine="openpyxl")
```

JSON

```
df = pd.read_json("data.json", lines=True)   # use lines=True for JSONL
```

Parquet (fast + compressed, great for big data)

```
df = pd.read_parquet("data.parquet")        # needs pyarrow or fastparquet
```

SQL (example with SQLite; similar for Postgres/MySQL)

```
import sqlite3
con = sqlite3.connect("warehouse.db")
df = pd.read_sql("SELECT order_id, date, country, amount FROM orders", con,
parse_dates=["date"])
con.close()
```

Many files → one DataFrame

```
from glob import glob
files = glob("logs/2025-07-*.csv")
```

df = pd.concat((pd.read_csv(f) for f in files), ignore_index=True)

---

2) QUICK INSPECT (understand raw data)

```
df.head()               # first 5 rows
df.tail(3)              # last 3 rows
df.shape                # (rows, cols)
df.info()               # dtypes + non-null counts
df.dtypes
df.sample(5, random_state=0) #no of rows, and seed

        df.describe(numeric_only=True)   # numeric stats
```

df["country"].value_counts(dropna=False) # **.value_counts():** Counts how many times each unique value appears in that column. Sorts the counts by default in descending order (most frequent first).

```
df.isna().sum()              # missing values per column
```

---

3) CLEAN (fix names, types, missing, duplicates, text, dates, outliers)

3.1 Rename columns (make them consistent)

```
df = df.rename(columns={"Order ID": "order_id", "Order Date": "order_date"})
# or:
df.columns = (df.columns
            .str.strip()
            .str.lower()
            .str.replace(r"[^\w]+", "_", regex=True))
```

3.2 Fix data types

df["order_date"] = pd.to_datetime(df["order_date"], errors="coerce", dayfirst=False)

df["amount"]   = pd.to_numeric(df["amount"], errors="coerce") #ll **force invalid values to become `NaN`**

df["country"]   = df["country"].astype("string")   # modern string dtype

df["segment"]   = df["segment"].astype("category")   # categories save memory

df['column_name'].unique()

👉 Returns a NumPy array of unique values.

df['column_name'].nunique() //👉 Returns the number of unique values.

Frequency of Each Value

```
df['column_name'].value_counts()
```

👉 Shows each distinct value with its count (sorted by default).

With Normalized Percentages

`df['column_name'].value_counts(normalize=True)`

👉 Shows relative frequencies (percentages).

If `df['device_type']` has values like `['router', 'camera', 'router', 'sensor']`

- `df['device_type'].unique()` → `array(['router','camera','sensor'], dtype=object)`
- `df['device_type'].nunique()` → `3`
- `df['device_type'].value_counts()`

Fill missing values using interpolation:

```
df = pd.DataFrame(data)

print("Before interpolation:")
print(df)

# Fill missing values using interpolation
df['A'] = df['A'].interpolate()

print("\nAfter interpolation:")
print(df)
```

```
Before interpolation:
     A
0  1.0
1  2.0
2  NaN
3  4.0
4  5.0

After interpolation:
     A
0  1.0
1  2.0
2  3.0   <-- filled by interpolation (avg of 2 and 4)
3  4.0
4  5.0
```

ion

3.3 Handle missing values (NaN)

```
df = df.dropna(subset=["order_id", "order_date"])          # must-have cols

df["amount"]   = df["amount"].fillna(df["amount"].median()) # numeric impute
df["country"]  = df["country"].fillna("Unknown")            # text impute

# interpolate time series:
df = df.sort_values("order_date")
df["amount"] = df["amount"].interpolate(method="time")
```

Suppose you have:

order_date	amount
2024-01-01	100
2024-01-02	NaN
2024-01-04	200

1. First, sort by `order_date`.
2. Then `interpolate(method="time")` looks at the gap:
   - From Jan 1 → Jan 4 = 3 days.
   - Missing value (Jan 2) is 1 day after Jan 1, so it gets **closer to 100**.
   - If Jan 3 was also missing, it would get a value between Jan 2 and Jan 4.

Result would be something like:

order_date	amount
2024-01-01	100.0
2024-01-02	133.3
2024-01-04	200.0

So the `NaN` got filled proportionally with respect to time.

3.4 Remove duplicates
df = df.drop_duplicates(subset=["order_id"])   # keep first by default

3.5 Clean text columns
s = df["customer_name"].astype("string")
df["customer_name"] = (s.str.strip()
                         .str.replace(r"\s+", " ", regex=True)
                         .str.title()) # converts to title case: John Smith (first letter uppercase, rest lowercase)

# standardize categorical spellings
df["country"] = df["country"].replace({
    "U.S.A": "USA",
 "United States": "USA",
 "us": "USA"
})


```
df["country"] = df["country"].replace({
    "U.S.A": "USA",
    "United States": "USA",
    "us": "USA"
})
```


3.6 Parse / split / combine columns

split "City, State" into two columns
df[["city", "state"]] = df["city_state"].str.split(",", n=1, expand=True)
df["state"] = df["state"].str.strip()

extract pattern with regex (e.g., 3 letters + 4 digits)
df["sku"] = df["raw"].str.extract(r"([A-Z]{3}\d{4})", expand=False)

combine
df["full_name"] = df["first"].str.cat(df["last"], sep=" ")

3.7 Dates & new features
df["year"] = df["order_date"].dt.year
df["month"] = df["order_date"].dt.month
df["dow"]   = df["order_date"].dt.day_name()

3.8 Outliers (simple IQR rule)
q1, q3 = df["amount"].quantile([0.25, 0.75])
iqr = q3 - q1
low, high = q1 - 1.5*iqr, q3 + 1.5*iqr
df = df[df["amount"].between(low, high)] # or clip with .clip(lower=low, upper=high)

```
[100, 120, 130, 125, 110, 115, 1000]
```

- Q1 ≈ 115, Q3 ≈ 125
- IQR = 10
- Bounds = [115 - 15, 125 + 15] = [100, 140]
- So values between **100 and 140** are kept.
- `1000` is outside, so it's dropped.

3.9 Validate assumptions (catch bad rows early)
assert df["order_id"].is_unique
assert df["amount"].ge(0).all()
assert df["country"].isin(["USA", "India", "UK", "Unknown"]).all()

Tip: If an assert fails, inspect the offenders:
df.loc[~df["amount"].ge(0)]

---

4) TRANSFORM (group, reshape, join)
4.1 Filter, select, sort
df = df.loc[df["country"].eq("India"), ["order_id", "order_date", "amount"]]

df = df.sort_values(["order_date", "amount"], ascending=[True, False])

`df.loc` is used for:

👉 **Label-based selection** of **rows and columns**.

The general syntax is:

```
df.loc[rows, columns]
```

4.2 New columns (vectorized)
df["tax"] = df["amount"] * 0.18
df = df.assign(net=lambda x: x["amount"] - x["tax"])

4.3 Grouping & aggregation

```
summary = (df
   .groupby(["year", "country"], as_index=False)
   .agg(
      orders=("order_id", "count"),
      revenue=("amount", "sum"),
      avg_order=("amount", "mean"),
   ))
```

4.4 Pivot / unpivot
# Pivot (wide)
wide = summary.pivot(index="year", columns="country", values="revenue")

# Melt (long)
long = wide.reset_index().melt(id_vars="year", var_name="country", value_name="revenue")

4.5 Merge / join tables
# left join to add lookup attributes
df = df.merge(products[["sku", "category"]], on="sku", how="left")

In [34]:
```
df.groupby('Company')
```
Out[34]:
```
<pandas.core.groupby.DataFrameGroupBy object at 0x113014128>
```
You can save this object as a new variable:

In [35]:
```
by_comp = df.groupby("Company")
```
And then call aggregate methods off the object:

In [36]:
```
by_comp.mean()
```
Out[36]:

	Sales
Company	
FB	296.5
GOOG	160.0
MSFT	232.0

---

5) LOAD (write/export data)
```
# CSV
df.to_csv("clean_orders.csv", index=False)

# Excel (one or many sheets)
with pd.ExcelWriter("report.xlsx", engine="openpyxl") as xls:
    df.to_excel(xls, sheet_name="Orders", index=False)
    summary.to_excel(xls, sheet_name="Summary", index=False)

# Parquet (best for big/analytics)
df.to_parquet("clean_orders.parquet", compression="snappy", index=False)

# JSON (records or lines for streaming)
df.to_json("orders.json", orient="records")
df.to_json("orders.jsonl", orient="records", lines=True)

# SQL
```

```
import sqlite3
con = sqlite3.connect("warehouse.db")
df.to_sql("clean_orders", con, if_exists="replace", index=False)
con.close()
```

---

6) End-to-end mini example (copy/paste)

This shows the whole flow on a tiny sample.

```
import pandas as pd
from io import StringIO

raw_csv = StringIO("""
Order ID,Order Date,Country,Amount,Customer Name,City_State
1001,2025-07-01,India,120.5,  alice  ,Delhi, Delhi
1002,2025/07/02,United States,1,234.00,BOB  ,San Francisco, CA
1003,2025-07-03,us, ,Charlie,New York, NY
1001,2025-07-01,India,120.5,alice,Delhi, Delhi
""".replace(",", ","))
# (Note: if your editor auto-formats commas, just load from a real CSV file.)

# --- EXTRACT ---
df = pd.read_csv(raw_csv, na_values=["", " "])

# --- INSPECT ---
print(df.head())
print(df.info())

# --- CLEAN ---
df = df.rename(columns=str.strip)
df.columns = (df.columns.str.lower().str.replace(r"[^\w]+", "_", regex=True))

# types
df["order_date"] = pd.to_datetime(df["order_date"], errors="coerce",
infer_datetime_format=True)
df["amount"]     = pd.to_numeric(df["amount"].astype("string").str.replace(",", ""), errors="coerce")
df["country"]    = df["country"].astype("string")

# text normalization
df["customer_name"] = (df["customer_name"].astype("string")
                       .str.strip()
                       .str.title())

# split city/state
df[["city", "state"]] = (df.pop("city_state")
```

```
                       .astype("string")
                       .str.split(",", n=1, expand=True))
df["city"]  = df["city"].str.strip()
df["state"] = df["state"].str.strip()

# drop duplicates, handle missing
df = df.drop_duplicates(subset=["order_id"])
df["amount"] = df["amount"].fillna(df["amount"].median())
df["country"] = df["country"].replace({"U.S.A": "USA", "us": "USA", "United States": "USA"})

# features
df["year"]  = df["order_date"].dt.year
df["month"] = df["order_date"].dt.month

# validate
assert df["order_id"].is_unique
assert df["amount"].ge(0).all()

# --- TRANSFORM ---
summary = (df.groupby(["year", "country"], as_index=False)
           .agg(orders=("order_id","count"), revenue=("amount","sum")))

# --- LOAD ---
df.to_csv("clean_orders.csv", index=False)
summary.to_parquet("revenue_by_year_country.parquet", index=False)

print("Clean rows:", len(df))
print(summary)
```

---

7) Common gotchas (and fixes)

- File not found: check os.getcwd() and use a correct relative/absolute path.

SettingWithCopyWarning: avoid chained indexing like df[df.a>0]["b"]=.... Use .loc:

```
df.loc[df["a"] > 0, "b"] = 1
```

-

Mixed types in a column: coerce then clean:

```
df["amount"] = pd.to_numeric(df["amount"], errors="coerce")
```

-
- Date parsing issues: pass dayfirst=True or a format="%d-%m-%Y".

---

8) Performance tips (when data grows)

- Read only what you need: usecols=..., nrows=....
- Specify dtype= at read time; use category for repeated strings.
- Prefer Parquet over CSV for speed and size.
- Use chunksize= for huge CSVs and process batch-by-batch.
- Avoid Python loops; use vectorized ops, map/replace, groupby, .assign.

0) Setup & quick tips

```
import pandas as pd
import os

print(os.getcwd())     # where your notebook is running
print(os.listdir())    # what files are here
pd.set_option("display.max_rows", 8)
pd.set_option("display.max_columns", 20)
```

Paths:

- "data.csv" → file in the current folder
- "folder/data.csv" → relative path
- "/absolute/path/data.csv" → absolute path (starts with / on Linux/Mac, with a drive like C:\ on Windows)

---

1) EXTRACT (read/import data)

CSV (most common)

```
df = pd.read_csv(
    "sales.csv",
    usecols=["order_id", "date", "country", "amount"],  # read only needed cols
    dtype={"order_id": "Int64", "country": "string"},    # initial dtypes
    parse_dates=["date"],                               # parse to datetime
    na_values=["", "NA", "null", "?"],                  # treat these as NaN
    thousands=",",                                      # "1,234" -> 1234
    encoding="utf-8",                                   # or "latin-1"
)
```

Large files (stream in chunks)

```
chunks = pd.read_csv("huge.csv", chunksize=100_000)
df = pd.concat(
    (c[c["amount"] > 0] for c in chunks),    # example pre-filter
```

```
    ignore_index=True
)
```

Excel

```
df = pd.read_excel("book.xlsx", sheet_name="Sheet1", engine="openpyxl")
```

JSON

```
df = pd.read_json("data.json", lines=True)   # use lines=True for JSONL
```

Parquet (fast + compressed, great for big data)

```
df = pd.read_parquet("data.parquet")        # needs pyarrow or fastparquet
```

SQL (example with SQLite; similar for Postgres/MySQL)

```
import sqlite3
con = sqlite3.connect("warehouse.db")
df = pd.read_sql("SELECT order_id, date, country, amount FROM orders", con,
parse_dates=["date"])
con.close()
```

Many files → one DataFrame

```
from glob import glob
files = glob("logs/2025-07-*.csv")
df = pd.concat((pd.read_csv(f) for f in files), ignore_index=True)
```

---

2) QUICK INSPECT (understand raw data)

```
df.head()               # first 5 rows
df.tail(3)              # last 3 rows
df.shape                # (rows, cols)
df.info()               # dtypes + non-null counts
df.dtypes
df.sample(5, random_state=0)

df.describe(numeric_only=True)   # numeric stats
df["country"].value_counts(dropna=False)
df.isna().sum()                  # missing values per column
```

---

3) CLEAN (fix names, types, missing, duplicates, text, dates, outliers)

3.1 Rename columns (make them consistent)

```
df = df.rename(columns={"Order ID": "order_id", "Order Date": "order_date"})
# or:
df.columns = (df.columns
              .str.strip()
```

```
            .str.lower()
            .str.replace(r"[^\w]+", "_", regex=True))
```

3.2 Fix data types

```
df["order_date"] = pd.to_datetime(df["order_date"], errors="coerce", dayfirst=False)
df["amount"]     = pd.to_numeric(df["amount"], errors="coerce")
df["country"]    = df["country"].astype("string")          # modern string dtype
df["segment"]    = df["segment"].astype("category")        # categories save memory
```

3.3 Handle missing values (NaN)

```
df = df.dropna(subset=["order_id", "order_date"])          # must-have cols

df["amount"]   = df["amount"].fillna(df["amount"].median()) # numeric impute
df["country"]  = df["country"].fillna("Unknown")            # text impute

# interpolate time series:
df = df.sort_values("order_date")
df["amount"] = df["amount"].interpolate(method="time")
```

3.4 Remove duplicates

```
df = df.drop_duplicates(subset=["order_id"])               # keep first by default
```

3.5 Clean text columns

```
s = df["customer_name"].astype("string")
df["customer_name"] = (s.str.strip()
                  .str.replace(r"\s+", " ", regex=True)
                  .str.title())

# standardize categorical spellings
df["country"] = df["country"].replace({
   "U.S.A": "USA", "United States": "USA", "us": "USA"
})
```

3.6 Parse / split / combine columns

```
# split "City, State" into two columns
df[["city", "state"]] = df["city_state"].str.split(",", n=1, expand=True)
df["state"] = df["state"].str.strip()

# extract pattern with regex (e.g., 3 letters + 4 digits)
df["sku"] = df["raw"].str.extract(r"([A-Z]{3}\d{4})", expand=False)

# combine
df["full_name"] = df["first"].str.cat(df["last"], sep=" ")
```

3.7 Dates & new features
```
df["year"]  = df["order_date"].dt.year
df["month"] = df["order_date"].dt.month
df["dow"]   = df["order_date"].dt.day_name()
```

3.8 Outliers (simple IQR rule)
```
q1, q3 = df["amount"].quantile([0.25, 0.75])
iqr = q3 - q1
low, high = q1 - 1.5*iqr, q3 + 1.5*iqr
df = df[df["amount"].between(low, high)]  # or clip with .clip(lower=low, upper=high)
```

3.9 Validate assumptions (catch bad rows early)
```
assert df["order_id"].is_unique
assert df["amount"].ge(0).all()
assert df["country"].isin(["USA", "India", "UK", "Unknown"]).all()
```

Tip: If an assert fails, inspect the offenders:
```
df.loc[~df["amount"].ge(0)]
```

---

4) TRANSFORM (group, reshape, join)
4.1 Filter, select, sort
```
df = df.loc[df["country"].eq("India"), ["order_id", "order_date", "amount"]]
df = df.sort_values(["order_date", "amount"], ascending=[True, False])
```

4.2 New columns (vectorized)
```
df["tax"] = df["amount"] * 0.18
df = df.assign(net=lambda x: x["amount"] - x["tax"])
```

4.3 Grouping & aggregation
```
summary = (df
    .groupby(["year", "country"], as_index=False)
    .agg(
        orders=("order_id", "count"),
        revenue=("amount", "sum"),
        avg_order=("amount", "mean"),
    ))
```

4.4 Pivot / unpivot
```
# Pivot (wide)
wide = summary.pivot(index="year", columns="country", values="revenue")

# Melt (long)
long = wide.reset_index().melt(id_vars="year", var_name="country", value_name="revenue")
```

4.5 Merge / join tables

```
# left join to add lookup attributes
df = df.merge(products[["sku", "category"]], on="sku", how="left")
```

---

5) LOAD (write/export data)

```
# CSV
df.to_csv("clean_orders.csv", index=False)

# Excel (one or many sheets)
with pd.ExcelWriter("report.xlsx", engine="openpyxl") as xls:
    df.to_excel(xls, sheet_name="Orders", index=False)
    summary.to_excel(xls, sheet_name="Summary", index=False)

# Parquet (best for big/analytics)
df.to_parquet("clean_orders.parquet", compression="snappy", index=False)

# JSON (records or lines for streaming)
df.to_json("orders.json", orient="records")
df.to_json("orders.jsonl", orient="records", lines=True)

# SQL
import sqlite3
con = sqlite3.connect("warehouse.db")
df.to_sql("clean_orders", con, if_exists="replace", index=False)
con.close()
```

---

6) End-to-end mini example (copy/paste)

This shows the whole flow on a tiny sample.

```
import pandas as pd
from io import StringIO

raw_csv = StringIO("""
Order ID,Order Date,Country,Amount,Customer Name,City_State
1001,2025-07-01,India,120.5,  alice  ,Delhi, Delhi
1002,2025/07/02,United States,1,234.00,BOB  ,San Francisco, CA
1003,2025-07-03,us, ,Charlie,New York, NY
1001,2025-07-01,India,120.5,alice,Delhi, Delhi
""".replace(",", ","))
# (Note: if your editor auto-formats commas, just load from a real CSV file.)

# --- EXTRACT ---
```

```
df = pd.read_csv(raw_csv, na_values=["", " "])

# --- INSPECT ---
print(df.head())
print(df.info())

# --- CLEAN ---
df = df.rename(columns=str.strip)
df.columns = (df.columns.str.lower().str.replace(r"[^\w]+", "_", regex=True))

# types
df["order_date"] = pd.to_datetime(df["order_date"], errors="coerce",
infer_datetime_format=True)
df["amount"]     = pd.to_numeric(df["amount"].astype("string").str.replace(",", ""), errors="coerce")
df["country"]    = df["country"].astype("string")

# text normalization
df["customer_name"] = (df["customer_name"].astype("string")
                      .str.strip()
                      .str.title())

# split city/state
df[["city", "state"]] = (df.pop("city_state")
                        .astype("string")
                        .str.split(",", n=1, expand=True))
df["city"]  = df["city"].str.strip()
df["state"] = df["state"].str.strip()

# drop duplicates, handle missing
df = df.drop_duplicates(subset=["order_id"])
df["amount"] = df["amount"].fillna(df["amount"].median())
df["country"] = df["country"].replace({"U.S.A": "USA", "us": "USA", "United States": "USA"})

# features
df["year"]  = df["order_date"].dt.year
df["month"] = df["order_date"].dt.month

# validate
assert df["order_id"].is_unique
assert df["amount"].ge(0).all()

# --- TRANSFORM ---
summary = (df.groupby(["year", "country"], as_index=False)
          .agg(orders=("order_id","count"), revenue=("amount","sum")))
```

# --- LOAD ---
df.to_csv("clean_orders.csv", index=False)
summary.to_parquet("revenue_by_year_country.parquet", index=False)

print("Clean rows:", len(df))
print(summary)

---

7) Common gotchas (and fixes)

- File not found: check os.getcwd() and use a correct relative/absolute path.

SettingWithCopyWarning: avoid chained indexing like df[df.a>0]["b"]=.... Use .loc:

```
df.loc[df["a"] > 0, "b"] = 1
```

-

Mixed types in a column: coerce then clean:

```
df["amount"] = pd.to_numeric(df["amount"], errors="coerce")
```

-
- Date parsing issues: pass dayfirst=True or a format="%d-%m-%Y".

---

8) Performance tips (when data grows)

- Read only what you need: usecols=..., nrows=....
- Specify dtype= at read time; use category for repeated strings.
- Prefer Parquet over CSV for speed and size.
- Use chunksize= for huge CSVs and process batch-by-batch.
- Avoid Python loops; use vectorized ops, map/replace, groupby, .assign.

🔹 Common Pandas Data Types (dtypes)

1. Numeric types

- int64, int32, int16, int8 → integers of different sizes
  (e.g., int8 = -128 to 127, saves memory if values are small)
- float64, float32, float16 → floating-point numbers
  (precision vs memory trade-off: float64 is most precise, float16 saves memory)

1. int64, int32, int16, int8

- **When to use:** For integer (whole number) data like counts, IDs, or quantities.
- **Example:** `df["age"] = df["age"].astype("int32")`
- **Tip:** Use smaller sizes (`int8`, `int16`) if numbers are small → saves memory.

- ⚠️ Be careful: if values exceed the range, you'll get overflow errors.

---

2. Boolean

- bool → stores True / False values efficiently
 (internally often stored as 1 byte)

---

3. Object

- object → usually means **strings** (but can technically store any Python object)
  - Flexible but **not memory efficient**.
  - Example: "apple", "banana", "cat".

---

4. String (new in Pandas 1.0+)

- string → dedicated string dtype (better than object for text).
  - Example: "Alice", "Bob".
  - Easier to apply .str methods.

---

5. Category

- category → stores text values as **integer codes** + a lookup table.
  - Example: "Small", "Medium", "Large" stored as 0,1,2 internally.
  - Saves **a lot of memory** when you have **repeated values**.

   - Best for things like gender, region, product type.

---

6. Datetime / Timedelta

- datetime64[ns] → timestamps (date + time).
   - Example: "2025-08-24 10:30:00".
- timedelta[ns] → difference between two dates/times.
   - Example: "5 days 02:00:00".

---

7. Other dtypes

- complex → complex numbers (rare in business data).
- Sparse → optimized for columns with many missing (NaN) values.
- Interval → ranges of values (like 0–10, 10–20).
- Mixed → if pandas can’t infer a single dtype (should usually be avoided).

🔹 Example

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "id": [1, 2, 3],                          # int64
    "price": [10.5, 20.7, 30.1],              # float64
    "is_active": [True, False, True],         # bool
    "name": ["Alice", "Bob", "Charlie"],      # object (string)
    "category": pd.Series(["A", "B", "A"], dtype="category"),  # category
    "date": pd.to_datetime(["2025-01-01", "2025-02-01", "2025-03-01"])  # datetime64
})

print(df.dtypes)
```

👉 Output:

```
id           int64
price        float64
is_active    bool
```

```
name          object
category    category
date    datetime64[ns]
dtype: object
```