CHEATSHEET SILO

Pandas Cheat Sheet

DataFrames, Functions & Practical Examples

```
import pandas as pd

df = pd.read_csv("data.csv")

summary = df.groupby("region")["sales"].sum()
```

PANDAS 3.X	PRINTABLE	COPY-READY

A focused reference for DataFrames, importing, cleaning, filtering, grouping, merging, reshaping, exporting, performance, and troubleshooting.

Free practical reference	Created by the CheatSheetSilo Editorial Team. Full interactive guide: cheatsheetsilo.com/pandas-cheat-sheet/

SECTION 01-02

Quick Reference and Setup

Start with the core objects, installation commands, and high-frequency operations used in everyday Pandas work.

Task	Syntax	Purpose
Import Pandas	`import pandas as pd`	Use the standard alias.
Read CSV	`pd.read_csv("data.csv")`	Load comma-separated data.
Read Excel	`pd.read_excel("data.xlsx", sheet_name="Sales")`	Load an Excel worksheet.
Preview rows	`df.head() / df.tail()`	Inspect the beginning or end.
Inspect structure	`df.info()`	Review dtypes and missing values.
Select columns	`df[["region", "sales"]]`	Return a DataFrame subset.
Filter rows	`df.loc[df["sales"] > 1000]`	Keep rows matching a condition.
Sort rows	`df.sort_values("sales", ascending=False)`	Order rows by values.
Group values	`df.groupby("region")["sales"].sum()`	Calculate totals per group.
Export CSV	`df.to_csv("output.csv", index=False)`	Save without an index column.

Installation

```
python -m pip install pandas
python -m pip install --upgrade pandas

# Optional Excel and Parquet support
python -m pip install openpyxl pyarrow
```

```
import pandas as pd

print(pd.__version__)
pd.show_versions()
```

Version note This reference targets Pandas 3.x. DataFrame.applymap() has been removed; use DataFrame.map(). Copy-on-Write is always active.

SECTION 03-04

Series, DataFrames and Creation

Understand the two primary Pandas data structures and common ways to construct reliable DataFrames.

Object or task	Syntax	Result
Series	`pd.Series([10, 20, 30], name="sales")`	One-dimensional labeled values.
DataFrame from dict	`pd.DataFrame({"product": ["A", "B"], "sales": [10, 20]})`	Columns built from equal-length lists.
From records	`pd.DataFrame(records)`	Rows built from dictionaries.
Custom index	`pd.DataFrame(data, index=["row_1", "row_2"])`	Named row labels.
Copy a frame	`df.copy()`	Independent DataFrame object.
Column labels	`df.columns`	Index containing column names.
Row labels	`df.index`	Index containing row names.
Dimensions	`df.shape`	Tuple of rows and columns.
Data types	`df.dtypes`	One dtype per column.

Create typed data

```
orders = pd.DataFrame({
    "order_id": pd.Series([101, 102], dtype="Int64"),
    "region": pd.Series(["East", "West"], dtype="string"),
    "sales": pd.Series([1250.0, 1480.5], dtype="Float64"),
})

orders = orders.set_index("order_id")
print(orders.dtypes)
```

Index alignment Pandas aligns Series and DataFrames by labels during arithmetic and assignment. Inspect indexes when results contain unexpected missing values.

SECTION 05-06

Import and Inspect Data

Control input columns and types, then profile the result before applying transformations.

Source or check	Syntax	Tip
CSV	`pd.read_csv("sales.csv", usecols=cols, parse_dates=["date"])`	Load only required columns.
Excel	`pd.read_excel("sales.xlsx", sheet_name="Data")`	Choose a worksheet.
JSON	`pd.read_json("sales.json", orient="records")`	Match the JSON orientation.
Parquet	`pd.read_parquet("sales.parquet", columns=cols)`	Preserve analytical dtypes.
SQL	`pd.read_sql_query("SELECT * FROM sales", connection)`	Use parameterized queries for values.
Overview	`df.info(memory_usage="deep")`	Review dtypes and memory.
Statistics	`df.describe(include="all")`	Summarize numeric and categorical columns.
Missing	`df.isna().sum()`	Count missing values by column.
Cardinality	`df.nunique(dropna=False)`	Count distinct values.

Import with explicit options

```
sales = pd.read_csv(
    "sales.csv",
    usecols=["order_id", "order_date", "region", "sales"],
    dtype={"order_id": "Int64", "region": "string", "sales": "Float64"},
    parse_dates=["order_date"],
    na_values=["", "NA", "N/A", "unknown"],
)

print(sales.head())
sales.info(memory_usage="deep")
```

Large files Use usecols to load less data and chunksize to process reducible operations without loading every row at once.

SECTION 07-09

Select, Filter, Sort and Rename

Use label-based or positional indexing, compose boolean masks, and keep labels consistent.

Task	Syntax	Notes
One column	`df["sales"]`	Returns a Series.
Several columns	`df[["region", "sales"]]`	Returns a DataFrame.
Label selection	`df.loc[mask, ["region", "sales"]]`	Select by labels and masks.
Position selection	`df.iloc[:5, :3]`	First five rows and three columns.
Fast scalar	`df.at[10, "sales"] / df.iat[0, 2]`	Access one value.
Multiple conditions	`(df["sales"] > 500) & df["region"].isin(["East", "West"])`	Use &, |, and ~.
Text filter	`df["name"].str.contains("pro", case=False, na=False)`	Vectorized string search.
Sort	`df.sort_values(["region", "sales"], ascending=[True, False])`	Multiple sort keys.
Rename	`df.rename(columns={"amt": "sales"})`	Return renamed labels.
Reset index	`df.reset_index(drop=True)`	Create a simple RangeIndex.

```
mask = (
    df["region"].isin(["East", "West"])
    & df["sales"].between(500, 5000, inclusive="both")
    & df["customer_name"].str.contains("studio", case=False, na=False)
)

result = (
    df.loc[mask, ["order_id", "region", "customer_name", "sales"]]
      .sort_values("sales", ascending=False)
      .reset_index(drop=True)
)
```

Boolean operators Use &, |, and ~ between Pandas conditions. Wrap each comparison in parentheses; Python and/or/not do not perform element-wise Series logic.

SECTION 10-11

Missing Values and Data Cleaning

Normalize missing markers, convert types safely, validate ranges, and preserve an audit trail.

Task	Syntax	Behavior
Detect missing	`df.isna() / df.notna()`	Boolean mask.
Drop required gaps	`df.dropna(subset=["order_id", "sales"])`	Remove unusable rows.
Fill by column	`df.fillna({"region": "Unknown", "sales": 0})`	Column-specific defaults.
Forward fill	`df.ffill() / df.bfill()`	Propagate nearby values.
Interpolate	`df["sales"].interpolate()`	Estimate numeric gaps.
Convert numeric	`pd.to_numeric(df["sales"], errors="coerce")`	Invalid values become missing.
Convert dates	`pd.to_datetime(df["date"], errors="coerce")`	Invalid dates become NaT.
Nullable types	`df.convert_dtypes()`	Prefer Pandas nullable dtypes.
Limit range	`df["discount"].clip(0, 100)`	Cap values to valid limits.

```
clean = df.copy()

clean = clean.replace({"": pd.NA, "N/A": pd.NA, "unknown": pd.NA})
clean["sales"] = pd.to_numeric(clean["sales"], errors="coerce")
clean["order_date"] = pd.to_datetime(clean["order_date"], errors="coerce")
clean["discount"] = clean["discount"].clip(lower=0, upper=100)

invalid = clean.loc[
    clean["order_id"].isna()
    | clean["order_date"].isna()
    | clean["sales"].isna()
]

clean = clean.dropna(subset=["order_id", "order_date", "sales"])
```

Do not hide quality problems Keep invalid rows in a separate audit DataFrame before dropping or replacing them. A missing-value strategy should follow the meaning of the data.

SECTION 12-13

String and Date-Time Operations

Clean text with the str accessor and use datetime-aware columns for filtering, grouping, and durations.

Task	Syntax	Result
Trim text	`df["name"].astype("string").str.strip()`	Remove edge whitespace.
Normalize case	`df["email"].str.strip().str.lower()`	Comparable email values.
Literal replace	`s.str.replace("-", " ", regex=False)`	Replace exact text.
Split	`df["name"].str.split(" ", n=1, expand=True)`	Create separate columns.
Extract	`df["code"].str.extract(r"(?P[A-Z]+)-(?P\d+)")`	Regex capture groups.
Parse date	`pd.to_datetime(df["date"], errors="coerce")`	Datetime dtype.
Calendar fields	`df["date"].dt.year / .dt.month / .dt.day_name()`	Date components.
Date range	`pd.date_range("2026-01-01", periods=7, freq="D")`	Regular timestamps.
Duration	`(df["end"] - df["start"]).dt.total_seconds()`	Elapsed seconds.
Time zones	`df["utc"].dt.tz_convert("Europe/Copenhagen")`	Convert aware timestamps.

```
df["email"] = (
    df["email"].astype("string").str.strip().str.lower()
)

df["order_date"] = pd.to_datetime(df["order_date"], errors="coerce", utc=True)
df = df.assign(
    order_year=df["order_date"].dt.year,
    order_month=df["order_date"].dt.month,
    weekday=df["order_date"].dt.day_name(),
)

recent = df.loc[
    df["order_date"].between("2026-01-01", "2026-04-01", inclusive="left")
]
```

Timezone rule Use tz_localize() to assign a timezone to naive timestamps. Use tz_convert() only after timestamps are timezone-aware.

SECTION 14

GroupBy and Aggregation

Use split-apply-combine patterns to calculate summaries and transform rows relative to their groups.

Task	Syntax	Result
Sum	`df.groupby("region")["sales"].sum()`	One total per region.
Multiple keys	`df.groupby(["region", "product"])["sales"].sum()`	One total per combination.
Named aggregation	`df.groupby("region").agg(total_sales=("sales", "sum"))`	Readable output names.
SQL-style result	`df.groupby("region", as_index=False).agg(...)`	Keys remain columns.
Count rows	`df.groupby("region").size()`	Includes rows with missing values elsewhere.
Unique count	`df.groupby("region")["customer_id"].nunique()`	Distinct customers.
Transform	`df.groupby("region")["sales"].transform("mean")`	Group value per original row.
Filter groups	`df.groupby("region").filter(lambda g: g["sales"].sum() >= 100000)`	Keep complete qualifying groups.

```
regional = (
    df.groupby("region", as_index=False, dropna=False)
      .agg(
          total_sales=("sales", "sum"),
          average_sale=("sales", "mean"),
          order_count=("order_id", "count"),
          unique_customers=("customer_id", "nunique"),
      )
      .sort_values("total_sales", ascending=False)
)

region_total = df.groupby("region")["sales"].transform("sum")
df["region_share"] = df["sales"].div(region_total)
```

Pandas 3 categories observed=True is the default for categorical groupers. Set observed=False to include categories not present in the data.

SECTION 15

Merge, Join and Concat

Combine related tables, validate key relationships, audit unmatched rows, and concatenate efficiently.

Operation	Syntax	Purpose
Inner merge	`left.merge(right, on="id", how="inner")`	Keys present on both sides.
Left merge	`left.merge(right, on="id", how="left")`	Keep every left row.
Outer merge	`left.merge(right, on="id", how="outer", indicator=True)`	Union plus source indicator.
Different keys	`left.merge(right, left_on="customer_id", right_on="id")`	Match different column names.
Index join	`left.join(right, how="left")`	Align by index.
Anti join	`left.merge(right, on="id", how="left_anti")`	Left keys missing on right (Pandas 3).
Append rows	`pd.concat(frames, ignore_index=True)`	Stack frames vertically.
Combine columns	`pd.concat([a, b], axis="columns", join="inner")`	Align frames by index.

```
combined = orders.merge(
    customers[["customer_id", "customer_name", "segment"]],
    on="customer_id",
    how="left",
    validate="many_to_one",
    indicator="merge_source",
    suffixes=("_order", "_customer"),
)

unmatched = combined.loc[combined["merge_source"] == "left_only"]

frames = [pd.read_csv(path) for path in csv_files]
all_rows = pd.concat(frames, ignore_index=True, sort=False)
```

Merge safety Duplicate keys on both sides can multiply rows. Use validate to enforce expected relationships. Pandas also matches null merge keys with other null keys.

SECTION 16

Pivot and Reshape

Switch between long and wide forms, aggregate pivot tables, expand list values, and create cross-tabulations.

Task	Syntax	Result
Long to wide	`df.pivot(index="date", columns="region", values="sales")`	Unique combinations required.
Pivot summary	`df.pivot_table(index="region", columns="product", values="sales", aggfunc="sum")`	Aggregates duplicates.
Wide to long	`df.melt(id_vars="product", var_name="month", value_name="sales")`	Column labels become rows.
Unstack	`s.unstack("product", fill_value=0)`	Index level becomes columns.
Stack	`df.stack()`	Column level becomes index.
Explode	`df.explode("tags", ignore_index=True)`	One row per list item.
Crosstab	`pd.crosstab(df["region"], df["status"], normalize="index")`	Frequency or percentage table.
Indicators	`pd.get_dummies(df, columns=["region"], dtype="int8")`	One column per category.

```
summary = df.pivot_table(
    values="sales",
    index="region",
    columns="product",
    aggfunc=["sum", "mean"],
    fill_value=0,
    margins=True,
    margins_name="Total",
)

long_sales = monthly_sales.melt(
    id_vars="product",
    value_vars=["jan", "feb", "mar"],
    var_name="month",
    value_name="sales",
)
```

Pivot choice pivot() reshapes without aggregation and raises when combinations repeat. pivot_table() handles duplicates through an explicit aggregation rule.

SECTION 17-18

Apply, Map, Transform and Duplicates

Choose functions by input shape and remove repeated records using explicit business keys.

Task	Syntax	Guidance
Map labels	`df["region"].map(region_map)`	Dictionary or scalar function.
Map cells	`numeric.map(lambda x: round(x, 2), na_action="ignore")`	DataFrame element-wise.
Apply columns	`df[["sales", "cost"]].apply("mean")`	Function receives each column.
Apply rows	`df.apply(classify_order, axis="columns")`	Function receives each row.
Transform	`df.groupby("region")["sales"].transform("mean")`	Preserve original alignment.
Mark duplicates	`df.duplicated(subset="order_id", keep=False)`	All repeated business keys.
Remove duplicates	`df.drop_duplicates(subset="order_id", keep="last")`	Keep final row in current order.
Index uniqueness	`df.index.is_unique`	Check duplicate labels.

```
region_map = {"N": "North", "S": "South", "E": "East", "W": "West"}
df["region_name"] = df["region_code"].map(region_map).fillna("Unknown")

df["difference_from_mean"] = (
    df["sales"] - df.groupby("region")["sales"].transform("mean")
)

latest = (
    df.sort_values("updated_at", kind="stable")
      .drop_duplicates(subset="customer_id", keep="last", ignore_index=True)
)
```

Prefer vectorization Built-in arithmetic, string, datetime, aggregation, and GroupBy methods are generally clearer and faster than row-wise apply().

SECTION 19

Export Data

Write clean outputs for spreadsheets, APIs, analytics, and databases while controlling indexes and formats.

Format	Syntax	Use case
CSV	`df.to_csv("output.csv", index=False)`	Universal tabular exchange.
Compressed CSV	`df.to_csv("output.csv.gz", index=False, compression="gzip")`	Smaller text output.
Excel	`df.to_excel("report.xlsx", sheet_name="Sales", index=False)`	Human-facing workbook.
Multiple sheets	`with pd.ExcelWriter("report.xlsx") as writer: ...`	Several reports in one workbook.
JSON records	`df.to_json("output.json", orient="records", date_format="iso")`	API-style records.
JSON Lines	`df.to_json("output.jsonl", orient="records", lines=True)`	Streaming workflows.
Parquet	`df.to_parquet("output.parquet", index=False)`	Typed analytical storage.
SQL	`df.to_sql("sales", connection, if_exists="append", index=False)`	Database table.

```
df.to_csv(
    "sales-report.csv",
    columns=["order_id", "order_date", "region", "sales"],
    index=False,
    encoding="utf-8",
    na_rep="",
    float_format="%.2f",
    date_format="%Y-%m-%d",
)

with pd.ExcelWriter("sales-workbook.xlsx") as writer:
    df.to_excel(writer, sheet_name="Transactions", index=False)
    regional_summary.to_excel(writer, sheet_name="Regional Summary", index=False)
```

Dependencies Excel output normally requires openpyxl or XlsxWriter. Parquet requires pyarrow or fastparquet. Validate important exports by reading them back.

SECTION 20

Practical Sales Analysis Workflow

A compact end-to-end pattern for importing, cleaning, deduplicating, aggregating, and exporting a report.

```
import pandas as pd

sales = pd.read_csv(
    "sales.csv",
    usecols=["order_id", "order_date", "region", "customer_id", "sales"],
)

sales["order_date"] = pd.to_datetime(sales["order_date"], errors="coerce")
sales["sales"] = pd.to_numeric(sales["sales"], errors="coerce")
sales["region"] = sales["region"].astype("string").str.strip().str.title()

sales = (
    sales.dropna(subset=["order_id", "order_date", "region", "sales"])
         .drop_duplicates(subset="order_id", keep="last")
         .loc[lambda frame: frame["sales"] >= 0]
)

regional_summary = (
    sales.groupby("region", as_index=False)
         .agg(
             total_sales=("sales", "sum"),
             average_sale=("sales", "mean"),
             order_count=("order_id", "nunique"),
             customer_count=("customer_id", "nunique"),
         )
         .sort_values("total_sales", ascending=False)
)

regional_summary["average_sale"] = regional_summary["average_sale"].round(2)
regional_summary.to_csv("regional-sales-summary.csv", index=False)
print(regional_summary)
```

Workflow checklist

- Keep raw input separate from cleaned and aggregated outputs.
- Convert data types before filtering or calculating.
- Audit invalid and duplicate rows before removing them.
- Validate merge relationships and unmatched keys.
- Export only the columns required by the downstream consumer.
- Read critical output files back and compare row counts, columns, and dtypes.

Reproducibility	Keep each transformation explicit and ordered. A clear pipeline is easier to test, repeat, review, and debug than a collection of manual edits.

SECTION 21

Performance and Best Practices

Start with vectorization and efficient I/O, measure memory, then optimize the real bottleneck.

Practice	Preferred pattern	Why
Vectorize	`df["margin"] = df["sales"] - df["cost"]`	Avoid Python calls per row.
Load less	`pd.read_csv(path, usecols=cols, dtype=dtypes)`	Reduce I/O and memory.
Measure	`df.memory_usage(deep=True)`	Find expensive columns.
Categories	`df["region"].astype("category")`	Efficient repeated labels.
Chunk files	`pd.read_csv(path, chunksize=100_000)`	Bound memory use.
Concat once	`pd.concat(frames, ignore_index=True)`	Avoid repeated copies.
Safe assignment	`df.loc[df["sales"] < 0, "sales"] = 0`	Single clear operation.
Row iteration	`df.itertuples(index=False)`	Use only when unavoidable.

```
memory = df.memory_usage(index=True, deep=True).sort_values(ascending=False)
print(memory)
print(f"Total MB: {memory.sum() / 1024**2:.2f}")

partial_totals = []
for chunk in pd.read_csv(
    "large-sales.csv",
    usecols=["region", "sales"],
    dtype={"region": "string", "sales": "float64"},
    chunksize=100_000,
):
    partial_totals.append(chunk.groupby("region")["sales"].sum())

regional_totals = pd.concat(partial_totals).groupby(level=0).sum()
```

Copy-on-Write Pandas 3 objects returned from indexing behave independently while copies are delayed internally. Chained assignment never updates the original object; use one loc assignment.

SECTION 22-23

Troubleshooting and FAQ

Use targeted checks to diagnose labels, types, boolean logic, assignment, pivots, and merges.

Problem	Fix	Reason
KeyError	`print(df.columns.tolist())`	Inspect exact labels and whitespace.
Ambiguous truth	`(df["sales"] > 0).any() / .all()`	Reduce a boolean Series.
Chained assignment	`df.loc[mask, "sales"] = 0`	Update original in one operation.
.dt error	`pd.to_datetime(df["date"], errors="coerce")`	Use datetime dtype.
.str error	`df["name"].astype("string").str.strip()`	Use string dtype.
Pivot duplicates	`pivot_table(..., aggfunc="sum")`	Define how repeats combine.
Merge mismatch	`Normalize both key dtypes`	Keys must be comparable.
Overlap	`suffixes=("_left", "_right")`	Disambiguate repeated columns.

```
df.columns = (
    df.columns.str.strip().str.lower().str.replace(" ", "_", regex=False)
)

required = {"order_id", "sales"}
missing = required - set(df.columns)
if missing:
    raise KeyError(f"Missing columns: {sorted(missing)}")

orders["customer_id"] = orders["customer_id"].astype("string").str.strip()
customers["customer_id"] = customers["customer_id"].astype("string").str.strip()

combined = orders.merge(
    customers,
    on="customer_id",
    how="left",
    validate="many_to_one",
    indicator=True,
)
```

Fast diagnosis Inspect column names, dtypes, shape, missing counts, unique-key counts, and a small sample of affected rows. Avoid printing an entire large DataFrame.

Continue with the interactive guide

The web version includes searchable sections, copy buttons, expanded explanations, official references, and future updates.

cheatsheetsilo.com/pandas-cheat-sheet/

Core workflow	Recommended order	Outcome
Understand	`inspect -> select -> filter`	Know the source data.
Clean	`normalize -> convert -> validate`	Reliable types and values.
Analyze	`group -> merge -> reshape`	Useful summaries.
Deliver	`audit -> export -> verify`	Trusted output.

About CheatSheetSilo Free practical references for programming, data, spreadsheets, productivity, and AI. No sign-up required.

Created by the CheatSheetSilo Editorial Team - September 2026