CHEAT SHEET PYTHON / DATA ANALYSIS / BEGINNER

Filter Rows in pandas

Filter pandas DataFrame rows safely and clearly using boolean masks, multiple conditions, membership tests, string/date filters, null checks, and query expressions.

Updated Oct 4, 2026 Online version →



PROBLEM

You need to keep only the rows that match a business condition:

- orders above a value
- records for specific countries
- rows between two dates
- customers with missing data
- statuses in an allowed list
- text containing a pattern
- rows matching several conditions at once

The challenge is not just syntax.

You want filtering that is:

- correct
- readable
- null-safe
- easy to debug
- safe to maintain in production code

QUICK ANSWER

Use boolean masks with `.loc[]`.

PYTHON

```python
filtered = df.loc[
    (df["status"] == "active")
    & (df["amount"] >= 100)
]
```

For membership:

PYTHON

```python
filtered = df.loc[
    df["country"].isin(["US", "CA", "MX"])
]
```

For dates:

```python
filtered = df.loc[
    df["event_time"].between(
        pd.Timestamp("2026-10-01"),
        pd.Timestamp("2026-10-31"),
        inclusive="both",
    )
]
```

Prefer `.loc[mask]` for production code because the intent is explicit and easy to extend.

Code / commands

1. Filter with one condition

```python
filtered = df.loc[df["amount"] > 100]
```

Use this for a simple numeric or equality condition.

Examples:

```python
active = df.loc[df["status"] == "active"]
failed = df.loc[df["status"] == "failed"]
large_orders = df.loc[df["amount"] >= 1000]
```

2. Filter with multiple AND conditions

Use `&` and wrap each comparison in parentheses.

```python
filtered = df.loc[
    (df["status"] == "active")
    & (df["amount"] >= 100)
    & (df["country"] == "US")
]
```

Do not use Python's `and` between Series.

Incorrect:

```python
df["status"] == "active" and df["amount"] >= 100
```

Correct:

```python
(df["status"] == "active") & (df["amount"] >= 100)
```

3. Filter with OR conditions

Use `|`.

```python
filtered = df.loc[
    (df["status"] == "failed")
    | (df["status"] == "cancelled")
]
```

For many equality options, `.isin()` is cleaner.

```python
filtered = df.loc[
    df["status"].isin(["failed", "cancelled"])
]
```

4. Combine AND and OR carefully

```python
filtered = df.loc[
    (df["country"] == "US")
    & (
        (df["status"] == "active")
        | (df["priority"] == "high")
    )
]
```

Parentheses matter.

Do not rely on operator precedence when the expression is business-critical.

5. Exclude rows with `~`

```python
filtered = df.loc[
    ~df["status"].isin(["cancelled", "deleted"])
]
```

This means:

> keep rows whose status is not in the list

For one value:

```python
filtered = df.loc[df["status"] != "cancelled"]
```

6. Filter with `.isin()`

```python
allowed_countries = ["US", "CA", "MX"]

filtered = df.loc[
    df["country"].isin(allowed_countries)
]
```

Use `.isin()` when you have a list, set, Series, or other collection of accepted values.

This is usually clearer than chaining many OR conditions.

7. Filter null and non-null rows

Null rows:

```python
missing_email = df.loc[
    df["email"].isna()
]
```

Non-null rows:

```python
has_email = df.loc[
    df["email"].notna()
]
```

Do not compare with `== None` or `== np.nan`.

Use pandas null methods.

8. Filter by numeric range

Use `.between()`.

```python
filtered = df.loc[
    df["amount"].between(100, 500, inclusive="both")
]
```

Equivalent:

```python
filtered = df.loc[
    (df["amount"] >= 100)
    & (df["amount"] <= 500)
]
```

`.between()` is often easier to read.

9. Filter dates correctly

First ensure the column is datetime:

```python
df["event_time"] = pd.to_datetime(
    df["event_time"],
    errors="coerce",
)
```

Then filter:

```python
start = pd.Timestamp("2026-10-01")
end = pd.Timestamp("2026-10-31")

filtered = df.loc[
    df["event_time"].between(
        start,
        end,
        inclusive="both",
    )
]
```

If timestamps include time-of-day, be careful with end-of-day logic.

For a half-open monthly interval, this is often safer:

```python
filtered = df.loc[
    (df["event_time"] >= pd.Timestamp("2026-10-01"))
    & (df["event_time"] < pd.Timestamp("2026-11-01"))
]
```

This avoids accidentally excluding rows later on October 31.

10. Filter text with `.str.contains()`

```python
filtered = df.loc[
    df["description"]
    .str.contains("error", case=False, na=False)
]
```

Important options:

- `case=False` → case-insensitive
- `na=False` → null values become `False`
- `regex=` → controls whether pattern is treated as regular expression

For literal text:

```python
filtered = df.loc[
    df["description"]
    .str.contains(
        "a+b",
        case=False,
        na=False,
        regex=False,
    )
]
```

Use `regex=False` when the search text should not be interpreted as a regular expression.

11. Filter text prefix or suffix

Starts with:

```python
filtered = df.loc[
    df["code"].str.startswith("ABC", na=False)
]
```

Ends with:

```python
filtered = df.loc[
    df["filename"].str.endswith(".csv", na=False)
]
```

12. Filter exact text after normalization

When source data may contain inconsistent casing or whitespace:

```python
normalized_status = (
    df["status"]
    .astype("string")
    .str.strip()
    .str.lower()
)

filtered = df.loc[
    normalized_status.eq("active")
]
```

This is safer than permanently changing the original column during initial analysis.

13. Filter with a reusable boolean mask

```python
mask = (
    (df["status"] == "active")
    & (df["amount"] > 100)
    & df["country"].isin(["US", "CA"])
)

filtered = df.loc[mask]
```

This is useful when:

- the condition is complex
- you want to inspect the mask
- you reuse the condition
- you want easier debugging

Check how many rows match:

```python
print(mask.sum())
```

14. Filter and select columns at the same time

```python
filtered = df.loc[
    df["status"] == "active",
    ["customer_id", "amount", "event_time"],
]
```

This can reduce memory and make downstream logic clearer.

15. Use `.query()` for readable expressions

```python
filtered = df.query(
    "status == 'active' and amount >= 100"
)
```

External variables use `@`:

```python
min_amount = 100

filtered = df.query(
    "status == 'active' and amount >= @min_amount"
)
```

`.query()` can be very readable for simple expressions.

Prefer boolean masks when:

- column names are awkward
- you need complex string methods
- you need detailed null handling
- the expression is easier to debug explicitly

16. Filter based on another column

```python
filtered = df.loc[
    df["actual_amount"] > df["expected_amount"]
]
```

Useful for row-level comparisons.

Example:

```python
late = df.loc[
    df["actual_date"] > df["expected_date"]
]
```

17. Filter groups based on aggregate logic

Suppose you want customers whose total revenue exceeds 10,000.

```python
customer_totals = (
    df.groupby("customer_id")["revenue"]
    .transform("sum")
)

filtered = df.loc[
    customer_totals > 10_000
]
```

Use `transform()` when you need group-level logic while preserving the original row grain.

18. Keep the top-N rows after filtering

```python
filtered = (
    df.loc[df["status"] == "active"]
    .nlargest(10, "amount")
)
```

For smallest:

```python
filtered = (
    df.loc[df["status"] == "active"]
    .nsmallest(10, "amount")
)
```

This is more direct than sorting the full DataFrame when you only need the top rows.

19. Filter duplicates by key

Keep only rows with duplicated business keys:

```python
duplicates = df.loc[
    df.duplicated(
        subset=["customer_id", "order_id"],
        keep=False,
    )
]
```

Use this for investigation before deciding how to deduplicate.

20. Filter rows that match values from another DataFrame

```python
valid_ids = lookup_df["customer_id"]

filtered = df.loc[
    df["customer_id"].isin(valid_ids)
]
```

For large relational matching with columns from both datasets, a merge is often more appropriate.

Use `.isin()` for simple semi-join-style filtering.

What the result means

A filter creates a boolean decision for every row.

For example:

```python
mask = df["amount"] > 100
```

produces values conceptually like:

```
True
False
True
False
...
```

Then:

```python
df.loc[mask]
```

keeps only the `True` rows.

Important behaviors:

Pattern	Meaning
`&`	element-wise AND
`	`	`
`~`	element-wise NOT
`.isin()`	membership test
`.isna()`	missing value
`.notna()`	non-missing value
`.between()`	inclusive/exclusive range test
`.str.contains()`	text match
`.eq()`	equality comparison
`.ne()`	not equal
`.gt()` / `.ge()`	greater than / greater or equal
`.lt()` / `.le()`	less than / less or equal

What to look for

Result count is unexpectedly zero

Check:

- dtype mismatch
- whitespace
- casing
- date parsing
- nulls
- wrong boolean grouping
- timezone mismatch

Example:

```python
print(df["status"].value_counts(dropna=False))
```

before assuming `"active"` exists exactly as written.

Result count is unexpectedly high

Check whether OR conditions are too broad.

Example:

```python
mask = (
    (df["country"] == "US")
    | (df["status"] == "active")
)
```

This keeps rows matching either condition.

If you intended both, use `&`.

Nulls disappear unexpectedly

Comparisons against null values usually evaluate to false-like results for filtering.

If nulls matter, handle them explicitly.

Example:

```python
mask = (
    df["amount"].isna()
    | (df["amount"] > 100)
)
```

Text filter misses obvious matches

Check:

- casing
- leading/trailing spaces
- regex interpretation
- null values

Use normalization or explicit options.

Date filter misses end-of-day rows

This happens when using midnight as the upper bound.

Prefer half-open ranges:

```python
start <= timestamp < next_period_start
```

for full-day/month windows.

PRODUCTION NOTES

Prefer `.loc[]` for explicit filtering

These are both valid:

```python
df[df["amount"] > 100]
```

```python
df.loc[df["amount"] > 100]
```

`.loc[]` is often clearer when you later add column selection or assignments.

Avoid chained assignment after filtering

Risky pattern:

```python
df[df["amount"] > 100]["flag"] = True
```

Use:

```python
df.loc[df["amount"] > 100, "flag"] = True
```

This is clearer and avoids chained-assignment ambiguity.

Filters scan the relevant columns

Boolean filtering is generally O(n) over the rows involved.

On very large DataFrames:

- filter early
- select only needed columns
- avoid repeated equivalent masks
- consider predicate pushdown at read time when supported
- consider a distributed engine if the dataset no longer fits comfortably in memory

Regex can be expensive

`str.contains()` defaults to regex behavior.

For literal matching, use:

```python
regex=False
```

This also avoids accidental regex semantics.

Timezones matter

Comparing timezone-aware and timezone-naive timestamps can fail or produce incorrect logic if handled inconsistently.

Normalize the time model before filtering.

Null semantics must be deliberate

Do not assume missing values should be excluded.

Business rules may require:

- keep nulls
- exclude nulls
- treat null as unknown
- route nulls for remediation

Encode that intent explicitly.

COMMON MISTAKES

1. Using `and` / `or` instead of `&` / `|`

Wrong:

```python
(df["a"] > 1) and (df["b"] < 5)
```

Correct:

```python
(df["a"] > 1) & (df["b"] < 5)
```

2. Forgetting parentheses

Wrong or ambiguous:

```python
df["a"] > 1 & df["b"] < 5
```

Correct:

```python
(df["a"] > 1) & (df["b"] < 5)
```

3. Comparing nulls with `== np.nan`

Use:

```python
df["col"].isna()
```

not:

```python
df["col"] == np.nan
```

4. Using regex accidentally

This:

```python
str.contains("a+b")
```

treats `+` as regex syntax.

Use:

```python
str.contains("a+b", regex=False)
```

for literal text.

5. Filtering dates while they are still strings

Lexical string order is not a reliable substitute for parsed datetime semantics.

Parse first.

6. Overwriting source columns during exploration

Build derived Series first when validating transformations.

Then mutate deliberately.

7. Filtering after expensive operations

If you can reduce the dataset earlier, do it.

Filtering before joins/aggregations can significantly reduce memory and work.

NEXT STEP

Use the filtering result to decide what comes next.

- **Need totals by group** → move to GroupBy and Aggregations.
- **Need data from another table/DataFrame** → use Join and Merge DataFrames.
- **Unexpected null behavior** → inspect missing-value semantics.
- **Date logic is complex** → use the Dates and Timestamps cheat sheet.
- **Filter result has duplicate keys** → inspect business grain and deduplication.
- **Filtering is slow on very large data** → reduce columns/rows earlier or move the operation closer to the data source.
- **The same filter is reused repeatedly** → build and name a reusable mask.

Related deep dives

- Related cheat sheet: Inspect a pandas DataFrame
- Related cheat sheet: GroupBy and Aggregations in pandas
- Related cheat sheet: Join and Merge DataFrames
- Related cheat sheet: Handle Missing Values in pandas
- Related cheat sheet: Work with Dates and Timestamps
- Related cheat sheet: Sort, Rank and Deduplicate Data
- Related cheat sheet: Debug Common pandas Problems