

Pandas Library

Aayush Bajaj

July 10, 2026

Contents

1	the model: alignment is the core idea	1
2	indexing: loc, iloc, and the trap	2
3	groupby: split, apply, combine	3
4	merge and join	4
5	reshape: pivot, melt, stack	4
6	time series	5
7	method chaining style	5
8	performance	6
9	see also	7

pandas is **numpy** with *labels*. a **Series** is a 1-d array married to an **index**; a **DataFrame** is a dict of such columns sharing one row index. the single organising idea — the one that explains both the magic and the bugs — is that **every operation aligns on labels first** and computes second. everything on this page runs against pandas 3.0 (outputs are real).

official docs: <https://pandas.pydata.org/docs/> — the user guide's '10 minutes to pandas' undersells itself by about three hours

1 the model: alignment is the core idea

positional arithmetic is what numpy already does; pandas instead matches *labels*, outer-joining the two indices and inserting **NaN** where either side is silent:

```
1 import pandas as pd
2
3 s1 = pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"])
4 s2 = pd.Series([10.0, 20.0, 30.0], index=["b", "c", "d"])
5 print(s1 + s2)
```

```
a      NaN
b     12.0
c     23.0
d       NaN
dtype: float64
```

nothing lined up by position: **b** met **b** and **c** met **c** regardless of where they sat, and

the unmatched labels surfaced as missing data instead of silently pairing wrong values. this is the entire value proposition — merge two months of data with different row orders, subtract a baseline keyed by id, divide by a per-group total: alignment does the bookkeeping that positional code gets subtly wrong. the price: an unexpected **NaN** usually means *your labels disagreed*, not that data vanished — check indices before reaching for `fillna`.

2 indexing: loc, iloc, and the trap

three access idioms, strictly separated:

- `df.loc[label_rows, label_cols]` — by **label** (slices are *inclusive* of the endpoint).
- `df.iloc[i, j]` — by **position**, numpy semantics.
- `df[mask] / df.loc[mask, cols]` — **boolean** filtering.

```
1 import warnings
2 import pandas as pd
3
4 df = pd.DataFrame({"x": [1, -2, 3], "y": [10, 20, 30]},
5                   index=["r0", "r1", "r2"])
6 print(df.loc["r1", "y"], df.iloc[1, 1])      # label vs position: same cell
7 print(df.loc[df.x > 0, "y"].tolist())        # boolean + column, one step
8
9 # the chained-assignment trap (pandas 3 = copy-on-write):
10 with warnings.catch_warnings(record=True) as w:
11     warnings.simplefilter("always")
12     df[df.x > 0]["y"] = 99                    # two steps: filter THEN assign
13     print("warned:", w[0].category.__name__)
14     print("y unchanged:", df.y.tolist())
15
16 df.loc[df.x > 0, "y"] = 99                    # one step: works
17 print("y now      :", df.y.tolist())
```

```
20 20
[10, 30]
warned: ChainedAssignmentError
y unchanged: [10, 20, 30]
y now      : [99, 20, 99]
```

the infamous `SettingWithCopyWarning` era ended with pandas 3: under copy-on-write, `df[mask]` *always* behaves as a copy, so chained assignment *never* writes through — you get a `ChainedAssignmentError` warning and no effect, instead of the old maybe-it-worked lottery. the rule was and remains: one indexing operation per assignment, `df.loc[rows, cols] = value`.

*pre-3.0 the chained form
sometimes mutated the original
depending on internal block
layout — a heisenbug factory,
now mercifully shot*

3 groupby: split, apply, combine

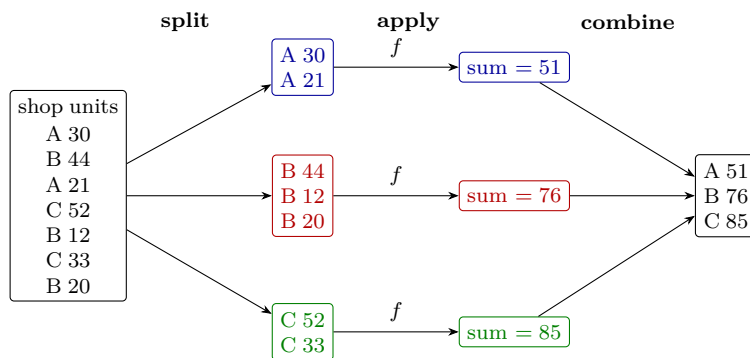


Figure 1: split-apply-combine: the frame is partitioned on the key, a function hits each part, and the parts are stitched back. `agg` shrinks each group to a row; `transform` returns a like-indexed frame; `filter` keeps or drops whole groups.

the three verbs differ in output shape, and choosing the wrong one is the classic `groupby` bug:

- **agg** — group → scalar(s). result has *one row per group*.
- **transform** — group → like-length column, broadcast back to the *original index* (the tool for “divide by group total” features).
- **filter** — group → boolean. keeps or drops *whole groups*, original shape otherwise untouched.

```

1 import numpy as np
2 import pandas as pd
3
4 sales = pd.DataFrame({
5     "shop": list("AABBBCC"),
6     "item": ["tea", "coffee", "tea", "coffee", "tea", "tea", "coffee"],
7     "units": [30, 21, 44, 12, 20, 52, 33],
8 })
9
10 g = sales.groupby("shop")["units"]
11 print(g.agg(["sum", "mean"]))           # agg: one row per group
12 sales["shop_share"] = g.transform(lambda u: u / u.sum()) # transform: same
13                                     # length
14 print(sales.head(4))
15 big = sales.groupby("shop").filter(lambda d: d.units.sum() > 60)
16 print("filter keeps shops:", sorted(big.shop.unique()))
  
```

	shop	sum	mean
A	51	25.500000	
B	76	25.333333	
C	85	42.500000	

	shop	item	units	shop_share
0	A	tea	30	0.588235
1	A	coffee	21	0.411765
2	B	tea	44	0.578947
3	B	coffee	12	0.157895

filter keeps shops: ['B', 'C']

4 merge and join

relational algebra over frames. `merge` is the general tool; the four joins differ only in which unmatched keys survive:

```
1 import pandas as pd
2
3 left = pd.DataFrame({"key": ["a", "b", "c"], "l": [1, 2, 3]})
4 right = pd.DataFrame({"key": ["b", "c", "d"], "r": [20, 30, 40]})
5
6 for how in ["inner", "left", "right", "outer"]:
7     m = left.merge(right, on="key", how=how)
8     print(f"{how:>6}: keys={m.key.tolist()}")
9
10 # validation + provenance
11 m = left.merge(right, on="key", how="outer", validate="one_to_one",
12               ↪ indicator=True)
13 print(m)
```

```
inner: keys=['b', 'c']
left: keys=['a', 'b', 'c']
right: keys=['b', 'c', 'd']
outer: keys=['a', 'b', 'c', 'd']
  key  l    r  _merge
0  a  1.0  NaN  left_only
1  b  2.0  20.0    both
2  c  3.0  30.0    both
3  d  NaN  40.0  right_only
```

two habits that pay for themselves: `validate="one_to_one"` (or `"one_to_many"` etc.) makes the merge *assert its own cardinality* — a silently duplicated key is the most expensive bug in data work, because row counts explode after the join and every downstream aggregate doubles; and `indicator=True` records where each row came from, which turns “why is this NaN” from archaeology into a groupby.

*the symptom is always
discovered three notebooks later,
in a plot that looks ‘a bit high’*

5 reshape: pivot, melt, stack

wide and tall are the two habitable phases of tabular data; these operators move between them:

```
1 import pandas as pd
2
3 sales = pd.DataFrame({
4     "shop": list("AABBBCC"),
5     "item": ["tea", "coffee", "tea", "coffee", "tea", "tea", "coffee"],
6     "units": [30, 21, 44, 12, 20, 52, 33],
7 })
8 wide = sales.pivot_table(index="shop", columns="item",
9                           values="units", aggfunc="sum", fill_value=0)
10 print(wide)
11 tall = wide.reset_index().melt(id_vars="shop", value_name="units")
12 print(tall.head(3))
13 print(wide.stack().head(4))    # columns -> inner index level
```

```
item  coffee  tea
shop
A      21    30
B      12    64
```

```

C          33   52
  shop    item  units
0   A  coffee    21
1   B  coffee    12
2   C  coffee    33
shop  item
A     coffee    21
      tea      30
B     coffee    12
      tea      64
dtype: int64

```

`pivot_table` = `groupby` + `reshape` (it aggregates duplicates; plain `pivot` refuses them). `melt` is its inverse, back to tidy one-observation-per-row form. `stack` and `unstack` are the same moves phrased as index-level rotation — a `pivot` is just `groupby` + `unstack` in a trenchcoat.

6 time series

the index earns its keep hardest when it is a `DatetimeIndex`: resampling is “groupby over time bins”, rolling is a sliding window with alignment handled for you.

```

1 import numpy as np
2 import pandas as pd
3 rng = np.random.default_rng(0)
4
5 idx = pd.date_range("2026-01-01", periods=90, freq="D")
6 ts = pd.Series(50 + np.arange(90) * 0.3 + rng.normal(0, 4, 90), index=idx)
7
8 print(ts.resample("MS").mean().round(2))           # downsample to month starts
9 roll = ts.rolling(window=7, center=True).mean()   # 7-day smoother
10 print("rolling covers", roll.notna().sum(), "of", len(ts), "days")
11 print("jan mean vs mar mean: %.1f -> %.1f" %
12       (ts["2026-01"].mean(), ts["2026-03"].mean()))

```

```

2026-01-01    53.90
2026-02-01    64.77
2026-03-01    72.79
Freq: MS, dtype: float64
rolling covers 84 of 90 days
jan mean vs mar mean: 53.9 -> 72.8

```

note the partial-string indexing (`ts["2026-01"]` selects the whole month) and the honest edges: a centred 7-day window cannot cover the first and last three days, and `rolling` says so with `NaN` rather than inventing data.

7 method chaining style

mutate-in-place scripts accumulate mystery state; the chaining idiom makes each frame a value and the pipeline a single readable expression — the fp instinct (see [functional programming](#)) applied to tables:

```

1 import numpy as np
2 import pandas as pd
3 rng = np.random.default_rng(0)
4

```

```

5 raw = pd.DataFrame({
6     "shop": rng.choice(["A", "B", "C"], 6),
7     "units": rng.integers(1, 50, 6),
8     "item": ["Tea ", "coffee", "TEA", " Coffee", "tea", "coffee "],
9 })
10
11 clean = (raw
12     .assign(item=lambda d: d.item.str.strip().str.lower())
13     .query("units > 5")
14     .pipe(lambda d: d.groupby(["shop", "item"], observed=True)
15         .units.sum()
16         .reset_index()))
17 print(clean)

```

	shop	item	units
0	A	coffee	85
1	A	tea	32
2	B	tea	9

`assign` adds derived columns without mutation, `query` filters with a readable mini-language, `pipe` slots any frame-to-frame function into the chain. debugging tip: comment out the tail of the chain and inspect the prefix — the pipeline structure makes bisection trivial.

8 performance

```

1 import numpy as np, pandas as pd, timeit
2 rng = np.random.default_rng(0)
3
4 n = 1_000_000
5 cities = pd.Series(rng.choice(["sydney", "melbourne", "brisbane"], n))
6 as_obj = cities.astype("object")
7 as_cat = cities.astype("category")
8 print("object strings: %5.1f MB" % (as_obj.memory_usage(deep=True) / 1e6))
9 print("category      : %5.1f MB" % (as_cat.memory_usage(deep=True) / 1e6))
10
11 t = lambda f: timeit.timeit(f, number=5) / 5 * 1e3
12 print("upper via .apply(str.upper): %6.1f ms" % t(lambda:
13     ↪ as_obj.apply(str.upper)))
14 print("upper via .str.upper()      : %6.1f ms" % t(lambda: as_obj.str.upper()))
15
16 # when pandas overhead dominates, drop to numpy
17 s = pd.Series(rng.standard_normal(n))
18 a = s.to_numpy() # zero-copy view of the same buffer
19 t_pd = timeit.timeit(lambda: (s - s.mean()) / s.std(), number=10) / 10
20 t_np = timeit.timeit(lambda: (a - a.mean()) / a.std(), number=10) / 10
21 print("standardise: pandas %5.2f ms | numpy %5.2f ms" % (t_pd * 1e3, t_np *
22     ↪ 1e3))

```

```

object strings: 56.7 MB
category      : 1.0 MB
upper via .apply(str.upper): 107.1 ms
upper via .str.upper()      : 73.5 ms
standardise: pandas 6.09 ms | numpy 2.60 ms

```

three honest lessons from the measurements:

- **categoricals** are the big win: a low-cardinality string column shrinks 57× (codes + a tiny dictionary), and groupbys on it get faster too. any id/label/state column

qualifies.

- **vectorised string ops** (`.str.*`) are tidier than `.apply` and comparable in speed (the ordering flips between runs — string work is loop-bound either way) — do not expect numpy-style $65\times$ miracles from text.
- **dropping to numpy** (`.to_numpy()`) is zero-copy for a single numeric dtype) roughly halves purely numerical pipelines by shedding index-alignment overhead. inside a tight loop, that factor is the difference between interactive and annoying.

9 see also

- [numpy](#) — the buffers underneath every column
- [matplotlib](#) — `df.plot()` is a thin wrapper over it
- [mastering pandas](#) — 100 exercises on the above
- [machine learning example with pandas](#) — a small end-to-end pipeline