

pandas Essentials

Data Manipulation in Python

Yen-Yi Ho

Department of Statistics, University of South Carolina

The Plan

Seven ideas. Everything else in pandas is a variation on these.

Idea
1 Series and DataFrame — labels travel with the data
2 Explore data — <code>info</code> , <code>describe</code> , <code>value_counts</code>
3 Selection — <code>loc</code> , <code>iloc</code> , and Boolean masks
4 New columns — <code>assign</code> and <code>map</code>
5 Missing data — <code>isna</code> , <code>dropna</code> , <code>fillna</code>
6 Split-apply-combine — <code>groupby</code>
7 Joining and reshaping — <code>merge</code> , <code>pivot_table</code>

The companion notebook has the same material with more examples and three exercises.

Why pandas?

pandas is the library for tabular data in Python — the `data.frame` of the Python world.

- Built on **NumPy**, but designed for **heterogeneous** tables
- Row and column **labels** travel with the data
- Operations **align on labels**, not on position
- First-class handling of **missing data**
- Reads CSV, Excel, SQL, JSON, Parquet, SAS, Stata

```
import numpy as np
import pandas as pd
```

Section 1

1. Series and DataFrame

The Series: A Labeled 1-D Array

```
s = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])
s
```

```
d    4
b    7
a   -5
c    3
dtype: int64
```

Two parts: the **values** and the **index**. The label is a meaningful handle on the value — unlike a plain NumPy array.

Labels Survive Everything

```
s[s > 0]
```

```
d      4  
b      7  
c      3  
dtype: int64
```

```
s * 2
```

```
d      8  
b     14  
a     -10  
c      6  
dtype: int64
```

Filtering and arithmetic both **preserve the index–value link**.

The One Behavior That Surprises Everyone

```
s2 = pd.Series([10, 20, 30], index=["a", "b", "z"])  
s + s2
```

```
a      5.0  
b     27.0  
c      NaN  
d      NaN  
z      NaN  
dtype: float64
```

Operations align on labels, not on position. Labels present in only one Series give NaN. This is pandas' most important behavior — and the source of most early confusion.

The DataFrame

A table: ordered **named columns**, each with its own dtype, sharing one **row index**. Think of it as a dictionary of Series.

```
df = pd.DataFrame({"state": ["Ohio", "Ohio", "Nevada"],  
                  "year": [2000, 2001, 2001],  
                  "pop": [1.5, 1.7, 2.4]})  
  
df
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Nevada	2001	2.4

Section 2

2. Explore Data

The First Things You Run

```
tips.shape
```

```
(244, 7)
```

```
tips.dtypes
```

```
total_bill    float64
```

```
tip           float64
```

```
tip_pct       float64
```

```
smoker        str
```

```
day           str
```

```
time          str
```

```
size          int64
```

```
dtype: object
```

shape for size, dtypes for types — a numeric column arriving as object is the most common data-loading defect, and this catches it in one line.

describe()

```
tips.describe().round(2)
```

	total_bill	tip	tip_pct	size
count	244.00	244.00	244.00	244.00
mean	23.22	4.11	0.18	2.59
std	8.00	1.98	0.06	0.92
min	7.77	0.98	0.08	1.00
25%	17.66	2.77	0.13	2.00
50%	21.93	3.78	0.18	2.00
75%	26.91	5.12	0.23	3.00
max	57.18	13.05	0.28	6.00

info()

```
tips.info()
```

```
<class 'pandas.DataFrame'>
```

```
RangeIndex: 244 entries, 0 to 243
```

```
Data columns (total 7 columns):
```

#	Column	Non-Null Count	Dtype
0	total_bill	244 non-null	float64
1	tip	244 non-null	float64
2	tip_pct	244 non-null	float64
3	smoker	244 non-null	str
4	day	244 non-null	str
5	time	244 non-null	str
6	size	244 non-null	int64

```
dtypes: float64(3), int64(1), str(3)
```

```
memory usage: 13.5 KB
```

Categories and Missing Values

```
tips["day"].value_counts()
```

```
day
Sat      84
Sun      75
Thur     59
Fri      26
Name: count, dtype: int64
```

```
tips.isna().sum().sum()      # total missing, whole frame
```

```
np.int64(0)
```

Run these five on every dataset, every time: `shape`, `dtypes`, `isna().sum()`, `duplicated().sum()`, and `value_counts()` on each category.

Which Is Which: Attributes and Methods

Attribute — no ()			Method — needs ()	
<code>.shape</code>	<code>.dtypes</code>		<code>.info()</code>	<code>.describe()</code>
<code>.columns</code>	<code>.index</code>		<code>.head()</code>	<code>.mean()</code>
<code>.values</code>	<code>.size</code>		<code>.sum()</code>	<code>.count()</code>
<code>.ndim</code>	<code>.empty</code>	<code>.T</code>	<code>.sort_values()</code>	<code>.groupby()</code>
<code>.loc</code>	<code>.iloc</code>	<code>.at</code>	<code>.isna()</code>	<code>.drop_duplicates()</code>

Attributes report the structure; methods do work on the data. Anything that takes arguments (`numeric_only=`, `axis=`) has to be a method.

Getting It Wrong

```
try:
    tips.shape()                # parentheses on an ATTRIBUTE
except TypeError as e:
    print("TypeError:", e)
```

TypeError: 'tuple' object is not callable

```
str(tips.info)[:38]            # parentheses MISSING on a method

'<bound method DataFrame.info of      t'
```

The second is the dangerous one: no error, no summary — just the function's description. `<bound method ...>` always means a forgotten `()`.

.loc Is an Attribute That Takes Brackets

```
tips.loc[0, "tip"]
```

```
np.float64(4.21)
```

.loc and .iloc look like exceptions, but they are attributes holding *indexer objects* — which is why they take **square** brackets rather than round ones.

And the trap in this dataset: `tips.size` is the attribute (244×7 elements), *not* the party-size column. Reach a column with `tips["size"]`, always.

Section 3

3. Selection

The Three Ways to Index

Syntax	Selects by	Typical use
<code>df["col"]</code>	column label	one column
<code>df[["a", "b"]]</code>	column labels	several columns
<code>df[mask]</code>	Boolean mask	filtering rows
<code>df.loc[rows, cols]</code>	labels	general, explicit
<code>df.iloc[rows, cols]</code>	integer positions	positional

Rule of thumb: use `loc` and `iloc` for anything beyond picking columns. Plain `[]` is ambiguous the moment the index is integer-valued.

loc and iloc

```
tips.loc[0:2, ["total_bill", "tip"]] # labels; endpoint INCLUDED
```

	total_bill	tip
0	23.57	4.21
1	17.18	4.03
2	17.40	3.62

```
tips.iloc[0:2, 0:2] # positions; endpoint EXCLUDED
```

	total_bill	tip
0	23.57	4.21
1	17.18	4.03

Boolean Filtering

```
big = tips["total_bill"] > 40  
tips.loc[big, ["total_bill", "tip", "day", "tip_pct"]].head(3)
```

	total_bill	tip	day	tip_pct
4	51.79	13.05	Thur	0.252
10	48.84	8.14	Thur	0.167
27	44.13	6.69	Thur	0.152

A Boolean Series is the most useful object in pandas: build it once, reuse it anywhere.

Combining Conditions

```
mask = (tips["total_bill"] > 30) & (tips["day"] == "Sun")  
mask.sum()
```

```
np.int64(11)
```

Use & (and), | (or), ~ (not) — **not** Python's and / or / not.

Parenthesize every condition. & binds more tightly than >, so without the parentheses Python evaluates the wrong thing and raises.

The Trap: Chained Assignment

```
# WRONG -- may modify a temporary copy and silently do nothing  
tips[tips["tip"] > 10]["tip_pct"] = 0.5
```

```
# RIGHT -- a single .loc call  
tips.loc[tips["tip"] > 10, "tip_pct"] = 0.5
```

If you ever see a `SettingWithCopyWarning`, you have written a chained assignment. Write one `loc` call instead.

Section 4

4. New Columns

Creating Columns

```
tips["bill_pp"] = tips["total_bill"] / tips["size"]  
tips[["total_bill", "size", "bill_pp"]].head(3)
```

	total_bill	size	bill_pp
0	23.57	4	5.893
1	17.18	2	8.590
2	17.40	2	8.700

Column arithmetic is vectorized — no loop, and the labels line up automatically.

Several at Once with assign

```
out = tips.assign(  
    is_weekend = lambda d: d["day"].isin(["Sat", "Sun"]),  
    bill_q = lambda d: pd.qcut(d["total_bill"], 4, labels=["Q1", "Q2", "Q3", "Q4"]))  
out[["day", "is_weekend", "bill_q"]].head(3)
```

	day	is_weekend	bill_q
0	Sat	True	Q3
1	Sun	True	Q1
2	Thur	False	Q1

assign returns a **new** frame, so it chains — and lambda d: lets each new column refer to the frame being built.

Recoding with map

```
kind = {"Thur": "Weekday", "Fri": "Weekday",  
        "Sat": "Weekend", "Sun": "Weekend"}  
tips["day"].map(kind).value_counts()
```

```
day  
Weekend    159  
Weekday     85  
Name: count, dtype: int64
```

map is the lookup workhorse — R's match-into-a-table idiom in one call.

Its one dangerous property: anything missing from the dictionary becomes NaN *silently*. Count the NaNs afterwards.

Section 5

5. Missing Data

Finding It

```
x = pd.Series([1.0, np.nan, 3.5, None, 7.0])  
x.isna()
```

```
0    False  
1     True  
2    False  
3     True  
4    False  
dtype: bool
```

Both `np.nan` and Python's `None` count as missing. For a frame, `df.isna().sum()` gives the count **per column** — run it on every new dataset.

Dropping and Filling

```
x.dropna().tolist()
```

```
[1.0, 3.5, 7.0]
```

```
[x.fillna(0).tolist(), x.ffill().tolist()]
```

```
[[1.0, 0.0, 3.5, 0.0, 7.0], [1.0, 1.0, 3.5, 3.5, 7.0]]
```

`dropna(how="all")`, `dropna(thresh=2)`, and `fillna({col: value})` cover most of the rest.

dropna on a Real Frame

```
d = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],  
                  [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])  
d.notna().sum(axis="columns")    # non-missing per ROW
```

```
0    3  
1    1  
2    0  
3    2  
dtype: int64
```

That count is what thresh is compared against.

Three Ways to Drop

```
d.dropna() # any missing -> drop
```

```
   0    1    2  
0  1.0  6.5  3.0
```

```
d.dropna(how="all") # all missing -> drop
```

```
   0    1    2  
0  1.0  6.5  3.0  
1  1.0  NaN  NaN  
3  NaN  6.5  3.0
```

```
d.dropna(thresh=2) # keep rows with >= 2 NON-missing
```

```
   0    1    2  
0  1.0  6.5  3.0  
3  NaN  6.5  3.0
```

Which Should You Use?

Option	When it is right
Drop the rows	few, and not systematic
Impute	many, and you must keep the sample size
Keep as its own category	the missingness is itself informative

Never impute an outcome variable — that invents the thing you are measuring.

And before you drop: look at **who** you are dropping. If the missing rows are all from one group, dropping them biases that group.

Section 6

6. Split–Apply–Combine

Split by a key $\Rightarrow$ **apply** a function to each piece $\Rightarrow$ **combine** the results.

```
tips.groupby("day")["tip_pct"].mean().round(4)
```

```
day
Fri      0.166
Sat      0.172
Sun      0.188
Thur     0.179
Name: tip_pct, dtype: float64
```

This is `group_by()` $\%>\%$ `summarize()` from `dplyr`, or `PROC MEANS BY` in SAS.

Several Keys

```
tips.groupby(["day", "smoker"])["tip_pct"].mean().round(3)
```

day	smoker	
Fri	No	0.164
	Yes	0.169
Sat	No	0.174
	Yes	0.169
Sun	No	0.186
	Yes	0.191
Thur	No	0.181
	Yes	0.177

Name: tip_pct, dtype: float64

Two keys give a **hierarchical index** — one row per combination.

unstack() Makes It Readable

```
tips.groupby(["day", "smoker"])["tip_pct"].mean().unstack().round(3)
```

smoker	No	Yes
day		
Fri	0.164	0.169
Sat	0.174	0.169
Sun	0.186	0.191
Thur	0.181	0.177

`unstack()` moves the inner level up into the columns. Same numbers, far easier to scan.

Named Aggregation

```
tips.groupby("smoker").agg(  
    n = ("tip", "size"),  
    avg_pct = ("tip_pct", "mean"),  
    max_bill = ("total_bill", "max")).round(3)
```

	n	avg_pct	max_bill
smoker			
No	143	0.178	57.18
Yes	101	0.177	51.79

Prefer this form. It names each output column, so you never end up parsing a hierarchical column index afterwards.

transform: One Value per Row, Not per Group

```
tips["day_mean"] = tips.groupby("day")["tip_pct"].transform("mean")
tips["vs_day"] = (tips["tip_pct"] - tips["day_mean"]) * 100
tips[["day", "tip_pct", "day_mean", "vs_day"]].head(3)
```

	day	tip_pct	day_mean	vs_day
0	Sat	0.179	0.172	0.750
1	Sun	0.235	0.188	4.687
2	Thur	0.208	0.179	2.853

agg returns **one row per group**; transform returns **one row per original row**, broadcasting the group result back.

That is exactly what you need for group-centering and within-group standardization.

Always Check Group Sizes

```
tips.groupby("day").size()
```

```
day
Fri      26
Sat      84
Sun      75
Thur     59
dtype: int64
```

A mean over 18 observations is not the same kind of number as a mean over 87. Report `n` beside every group statistic — `size()` in the `agg` call does it for free.

Section 7

7. Joining and Reshaping

merge

```
staff = pd.DataFrame({"day": ["Thur", "Fri", "Sat", "Sun"],  
                      "on_shift": [4, 5, 9, 8]})  
tips.merge(staff, on="day", how="left")[["day", "tip", "on_shift"]].head(3)
```

	day	tip	on_shift
0	Sat	4.21	9
1	Sun	4.03	8
2	Thur	3.62	4

how="inner"	"left"	"outer"
in both	all of the left	union

The Merge That Eats Your Data

```
before = len(df)
out = df.merge(lookup, on="key", how="left")
assert len(out) == before          # cheapest insurance you will ever write
```

If the right table has duplicate keys, a merge silently adds rows. A 3-row frame joined to a table with two matching keys comes back with 4 rows, and nothing warns you.

Check `lookup["key"].is_unique` first, or pass `validate="m:1"` and let pandas refuse.

pivot_table and crosstab

```
tips.pivot_table(index="day", columns="smoker",  
                  values="tip_pct", aggfunc="mean").round(3)
```

smoker	No	Yes
day		
Fri	0.164	0.169
Sat	0.174	0.169
Sun	0.186	0.191
Thur	0.181	0.177

```
pd.crosstab(tips["day"], tips["smoker"]).head(2)
```

smoker	No	Yes
day		
Fri	15	11
Sat	45	39

`pivot_table` is `groupby` + `unstack` in one call. `crosstab` is the counting special case — R's `table()`.

Method Chaining

```
(tips
 .query("size >= 2")
 .groupby("day")
 .agg(n=("tip", "size"), avg=("tip_pct", "mean"))
 .sort_values("avg", ascending=False)
 .round(3))
```

	n	avg
day		
Sun	74	0.189
Thur	59	0.179
Fri	24	0.172
Sat	84	0.172

Wrap in parentheses, one step per line — pandas' answer to the `|>` pipe.

Section 8

Wrapping Up

pandas and dplyr Side by Side

dplyr / tidyr	pandas
<code>filter(df, x > 1)</code>	<code>df[df["x"] > 1]</code>
<code>select(df, a, b)</code>	<code>df[["a", "b"]]</code>
<code>mutate(df, z = x + y)</code>	<code>df.assign(z=df["x"] + df["y"])</code>
<code>arrange(df, desc(x))</code>	<code>df.sort_values("x", ascending=False)</code>
<code>group_by(g) %>% summarize(m = mean(x))</code>	<code>df.groupby("g")["x"].mean()</code>
<code>left_join(a, b, by = "k")</code>	<code>pd.merge(a, b, on="k", how="left")</code>
<code>distinct(df)</code>	<code>df.drop_duplicates()</code>
<code>count(df, g)</code>	<code>df["g"].value_counts()</code>
<code>x %in% y</code>	<code>s.isin(y)</code>

The Five Silent Failures

None of these raise an error. Your code runs; your numbers are wrong.

- ❶ **Alignment.** Operations match on *labels*, not position. Two Series with different indexes give you NaN, not an error.
- ❷ **Chained assignment.** `df[mask]["col"] = x` may do nothing. Write one `.loc` call.
- ❸ **and / or.** Use `&`, `|`, `~`, and parenthesize each condition.
- ❹ **Silent NaN from map.** A key missing from the dictionary becomes missing data with no warning. Count afterwards.
- ❺ **Merges that grow.** Duplicate keys multiply rows silently. Check `shape` before and after, every time.

Exercises

Using the tips DataFrame:

- 1 Select the checks where `size` is at least 4 **and** `time` is "Dinner". How many are there, and what is their mean `tip_pct`?
- 2 Add a column `bill_pp = total_bill / size`. Then report, for each day, the number of checks and the mean `bill_pp` — in a single `groupby().agg()` call with named outputs.
- 3 Build a lookup table mapping each day to "Weekend" or "Weekday". Attach it to `tips` **twice** — once with `map`, once with `merge` — and confirm that the row count did not change.

Exercise 1 — One Solution

```
(tips
 .query("size >= 4")
 .groupby("time")
 .agg(n=("tip", "size"), avg=("tip_pct", "mean")).round(3))
```

	n	avg
time		
Dinner	34	0.187
Lunch	16	0.159

Grouping by time answers the question *and* shows the Lunch comparison for free.

Exercise 2 — One Solution

```
(tips
 .assign(bill_pp=lambda d: d["total_bill"] / d["size"])
 .groupby("day")
 .agg(n=("tip", "size"), avg_pp=("bill_pp", "mean")).round(3))
```

	n	avg_pp
day		
Fri	26	9.772
Sat	84	9.400
Sun	75	10.221
Thur	59	10.392

assign inside the chain, so bill_pp exists by the time agg needs it.

Exercise 3 — One Solution

```
look_up = pd.DataFrame({"day": ["Thur", "Fri", "Sat", "Sun"],  
                        "isWeekend": [False, False, True, True]})  
key = look_up.set_index("day")["isWeekend"]
```

```
print("rows before:", len(tips))  
tips["isWeekend"] = tips["day"].map(key)  
print("rows after :", len(tips))
```

rows before: 244

rows after : 244

Exercise 3 — The merge Version

```
merged = tips.merge(look_up, on="day", how="left")
print("rows:", len(tips), "->", len(merged))
merged.columns.tolist()[-3:]
```

```
rows: 244 -> 244
```

```
['vs_day', 'isWeekend_x', 'isWeekend_y']
```

Note the `_x` / `_y` suffixes. `tips` already had an `isWeekend` column from the `map` step, so `merge` kept both and renamed them rather than overwriting. Pass `suffixes=` to control the names — or drop the old column first.

Where to Go Next

- McKinney, W. *Python for Data Analysis*, 3rd ed. — <https://wesmckinney.com/book/>
 - Ch. 5 Getting started with pandas Ch. 7 Data cleaning
 - Ch. 8 Join, combine, reshape Ch. 10 Group operations
- VanderPlas, J. *Python Data Science Handbook*, Ch. 3 — <https://jakevdp.github.io/PythonDataScienceHandbook/>
- pandas user guide — https://pandas.pydata.org/docs/user_guide/
- Comparison with R — https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_r.html