

A Data Analysis with pandas

From a Messy File to a Defensible Finding

Yen-Yi Ho

Department of Statistics, University of South Carolina

Section 1

Stage 0: The Analysis Task

The Brief

A restaurant owner hands you one file and three questions.

“Every server believes something different about who tips well. I want to know what the data actually says.”

- 1 What is a **typical** tip at this restaurant? Does the “15 to 20 percent” norm hold?
- 2 Does the tip rate depend on the **size of the bill**?
- 3 Do **smokers, men, women, weekends** — any of the groups servers argue about — tip differently?

Deliverable: a short written finding for each question, with the evidence behind it and an honest statement of what the data *cannot* settle.

The Data

`tips_messy.csv` — one row per check, as exported from the restaurant's register.

Column	Meaning	Type we expect
<code>check_id</code>	identifier for the check	text
<code>total_bill</code>	amount of the bill, dollars	number
<code>tip</code>	tip left, dollars	number
<code>sex</code>	sex of the person paying	category
<code>smoker</code>	was the party in the smoking section	category
<code>day</code>	day of the week	category
<code>time</code>	lunch or dinner	category
<code>size</code>	number of people in the party	integer

Real data collected by a waiter over about ten weeks, reported in Bryant & Smith (1995).

The Workflow

Every analysis you will ever do has the same spine. We will walk all eight stages.

Stage		The question it answers
1	Load and inspect	What did I actually get?
2	Clean	What is wrong with it, and what will I do?
3	Derive	What do I need that is not in the file?
4	One variable	How does the outcome behave on its own?
5	Two variables	What moves with what?
6	Compare groups	Is the difference real, or is it something else?
7	Visualize	What is the one picture that shows it?
8	Report	What can I claim, and what can I not?

Stages 1–3 take most of the time. That is normal, and it is not wasted time.

Two Ground Rules Before We Start

1. **Never modify the raw file.** Read it, copy it, transform the copy. The raw file is your evidence; if you overwrite it you cannot reproduce or check anything.
2. **Every cleaning decision is a research decision.** Dropping a row, recoding a label, filling a missing value — each changes your answer. Write each one down as you make it, because you will have to defend it.

```
raw = pd.read_csv("tips_messy.csv")    # read once  
tips = raw.copy()                     # work on a copy
```

Section 2

Stage 1: Load and Inspect

Reading the File

```
raw = pd.read_csv("tips_messy.csv")  
raw.shape
```

(247, 9)

247 rows, 9 columns. The owner said there were “about 240 checks,” so the row count is plausible — but 9 columns is one more than the 8 we were promised.

Always check shape first. If it surprises you, stop and find out why before doing anything else.

First Look

```
raw.head(4)
```

	check_id	total_bill	tip	sex	smoker	day	time	size	\
0	C0004	\$23.68	3.31	Male	No	Sun	Dinner	2.0	
1	C0083	\$10.07	1.83	Female	No	Thur	lunch	1.0	
2	C0241	\$27.18	2.00	Female	Yes	Sat	dinner	2.0	
3	C0081	\$19.44	3.00	Male	Yes	THUR	Lunch	2.0	

	server_note
0	NaN
1	NaN
2	NaN
3	NaN

Already visible: a \$ in total_bill, a stray server_note column, and inconsistent capitalization in day and time.

info(): The Most Informative Single Call

```
raw.info()
```

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

```
RangeIndex: 247 entries, 0 to 246
```

```
Data columns (total 9 columns):
```

#	Column	Non-Null Count	Dtype
0	check_id	247 non-null	object
1	total_bill	247 non-null	object
2	tip	244 non-null	float64
3	sex	247 non-null	object
4	smoker	247 non-null	object
5	day	247 non-null	object
6	time	247 non-null	object
7	size	243 non-null	float64
8	server_note	5 non-null	object

```
dtypes: float64(2), object(7)
```

```
memory usage: 17.5+ KB
```

What dtypes Is Telling You

```
raw.dtypes
```

```
check_id      object
total_bill    object
tip           float64
sex           object
...
day           object
time         object
size         float64
server_note   object
Length: 9, dtype: object
```

total_bill is text, not a number. pandas read it as object/str because of the dollar signs — so `raw["total_bill"].mean()` would fail, or worse, concatenate.

A numeric column arriving as text is the single most common data-loading defect. `dtypes` catches it in one line.

describe() on What Little Is Numeric

```
raw.describe()
```

	tip	size
count	244.000	243.000
mean	1.771	2.576
std	11.350	0.956
min	-99.000	1.000
25%	2.000	2.000
50%	2.960	2.000
75%	3.603	3.000
max	10.000	6.000

Only two columns are numeric, and tip has a **minimum of** -99 — impossible for a tip. That is a *sentinel value*: a code someone used to mean “missing.”

Inspecting the Categorical Columns

```
raw["sex"].value_counts()
```

```
sex
Male      116
Female    62
M         18
F         18
male      16
female     9
MALE       8
Name: count, dtype: int64
```

Seven spellings for two categories. Any `groupby("sex")` right now would produce seven groups and a meaningless answer.

And the Rest

```
raw["day"].value_counts().head(8)
```

```
day
Sat      65
Sun      56
Thur     44
Fri      15
SUN      10
  Sat      8
  thur      6
  Thur      6
Name: count, dtype: int64
```

Twenty spellings for four days: case differences plus leading and trailing spaces.

Counting Missing Values and Duplicates

```
raw.isna().sum()
```

```
check_id      0
total_bill    0
tip           3
sex           0
...
day           0
time         0
size         4
server_note   242
Length: 9, dtype: int64
```

```
raw.duplicated().sum()
```

```
np.int64(3)
```

The Five Questions to Ask Any New Dataset

Question	One line that answers it
1 How big is it?	<code>df.shape</code>
2 What type is each column?	<code>df.dtypes / df.info()</code>
3 What is missing?	<code>df.isna().sum()</code>
4 Are there repeated rows?	<code>df.duplicated().sum()</code>
5 What values does each category take?	<code>df[c].value_counts()</code>

Run all five on **every** dataset, every time, before you analyze anything. They take thirty seconds and they will save you hours.

Our Defect Inventory

#	Defect	Fix
1	<code>server_note</code> is empty and unused	drop the column
2	<code>total_bill</code> is text with \$	strip, convert to float
3	<code>tip</code> uses <code>-99</code> for missing	replace with <code>NaN</code>
4	<code>sex</code> , <code>smoker</code> have many spellings	harmonize
5	<code>day</code> , <code>time</code> have case/space noise	strip, title-case
6	3 duplicated rows	<code>drop_duplicates</code>
7	6 missing tips, 4 missing sizes	decide, then document

Write the inventory down before you write any cleaning code. It becomes your checklist, and later it becomes a paragraph of your report.

Section 3

Stage 2: Clean

Start From a Copy

```
d = raw.copy()  
d.shape
```

```
(247, 9)
```

From here on, `raw` is untouched evidence and `d` is the working copy. If a cleaning step goes wrong you re-run from `raw` — you never re-download.

Defect 1: A Column Nobody Needs

```
raw["server_note"].value_counts(dropna=False)
```

```
server_note  
NaN      242  
comp      5  
Name: count, dtype: int64
```

```
d = d.drop(columns="server_note")
```

242 blanks and 5 comp flags. Not enough to analyze, so we drop it — and we say we dropped it in the report. Silently discarding a column is how findings become irreproducible.

Defect 2: Money Stored as Text

```
raw["total_bill"].head(3).tolist()
```

```
['$23.68', '$10.07', '$27.18']
```

```
d["total_bill"] = (d["total_bill"]  
                    .str.strip()                                # kill the padding  
                    .str.replace(r"$,)", "", regex=True)        # kill $ and ,  
                    .astype(float))
```

```
d["total_bill"].head(3).tolist()
```

```
[23.68, 10.07, 27.18]
```

Chain `.str` methods, then `astype`. Removing `,` matters the moment a bill reaches \$1,000.

Did the Conversion Lose Anything?

```
d["total_bill"].isna().sum(), d["total_bill"].min(), d["total_bill"].max()  
  
(np.int64(0), 3.07, 50.81)
```

Always check for new missing values after a type conversion. If a single value had failed to parse, `astype(float)` would have raised — but `pd.to_numeric(..., errors="coerce")` turns failures into NaN *silently*, which is why you count them afterwards.

A range of \$3.07 to \$50.81 for restaurant checks is plausible. Good.

Defect 3: A Sentinel Value

```
(raw["tip"] == -99).sum()
```

```
np.int64(3)
```

```
d["tip"] = d["tip"].replace(-99, np.nan)  
d["tip"].min()
```

```
1.0
```

−99, −999, 9999, "N/A", ".", −1 — every institution has its own code for “missing,” and none of them are missing to pandas until you say so. `describe()` catches them, because an impossible minimum or maximum sticks out.

Defect 4: One Category, Many Spellings

```
sorted(raw["smoker"].unique())
```

```
['N', 'No', 'Y', 'Yes', 'no', 'yes']
```

You *could* write a dictionary with every spelling in it. That works until the next export contains "YES " and quietly produces a NaN.

Better: **normalize first, then map**. Strip the whitespace, upper-case it, take the first letter — now there are only two possible values left to map.

The Robust Fix

```
d["sex"]      = d["sex"].str.strip().str.upper().str[0].map({"M": "Male", "F": "Female"})
d["smoker"]   = d["smoker"].str.strip().str.upper().str[0].map({"N": "No", "Y": "Yes"})
d["day"]      = d["day"].str.strip().str.title()
d["time"]     = d["time"].str.strip().str.title()
```

```
[sorted(d[c].unique()) for c in ["sex", "smoker", "day", "time"]]
```

```
[['Female', 'Male'],
 ['No', 'Yes'],
 ['Fri', 'Sat', 'Sun', 'Thur'],
 ['Dinner', 'Lunch']]
```

Four clean categoricals. `str.title()` handles "SUN", "sun", and " Sun " alike.

Always Check the Map Caught Everything

```
d[["sex", "smoker"]].isna().sum()
```

```
sex      0  
smoker   0  
dtype: int64
```

‘map’ returns ‘NaN’ for anything not in the dictionary — silently. That is its one dangerous property. A spelling you did not anticipate does not raise an error; it becomes a missing value and vanishes from your groupby.

So after every map on a category, count the NaNs. Zero here, so nothing escaped.

Defect 5: Duplicate Rows

```
d[d.duplicated(keep=False)].sort_values("check_id").head(4)
```

	check_id	total_bill	tip	sex	smoker	day	time	size
20	C0012	35.26	5.0	Female	No	Sun	Dinner	4.0
48	C0012	35.26	5.0	Female	No	Sun	Dinner	4.0
28	C0091	28.97	3.0	Male	Yes	Fri	Dinner	2.0
53	C0091	28.97	3.0	Male	Yes	Fri	Dinner	2.0

Dropping Them

```
before = len(d)
d = d.drop_duplicates()
before, len(d)
```

(247, 244)

Three exact duplicates removed. These are *exact* copies including `check_id`, so they are an export artifact rather than two real checks.

Be careful: two different checks could legitimately have identical bills and tips. Here the repeated `check_id` is what proves they are duplicates. Without an ID, `drop_duplicates()` on the full row would destroy real data.

Defect 6: Missing Values — Count First

```
d.isna().sum()
```

```
check_id      0
total_bill     0
tip           6
sex           0
smoker        0
day           0
time          0
size          4
dtype: int64
```

```
round(100 * d[["tip", "size"]].isna().mean(), 1)
```

```
tip      2.5
size     1.6
dtype: float64
```

About 2.5% of tips and 1.7% of sizes. Small — but “small” is not a decision.

Missing Values — Then Decide

Three options, and the choice depends on what the variable does.

Option	When it is right	Here?
Drop the rows	few, and not systematic	yes
Impute	many, and you must keep sample size	no
Keep as a category	missingness is itself informative	no

tip is our **outcome variable**. Imputing an outcome means inventing the very thing we are trying to measure, so that is out. With under 3% missing, we drop — and we report that we dropped 10 rows.

Check Before You Drop

```
gone = d[d[["tip", "size"]].isna().any(axis="columns")]
gone["day"].value_counts()
```

```
day
Sat      5
Sun      3
Thur     1
Fri      1
Name: count, dtype: int64
```

The question that matters: are the dropped rows unusual? If every missing tip were on a Saturday, dropping them would bias the Saturday estimate. Spread across all four days, so dropping is defensible.

Doing the Drop

```
d = d.dropna(subset=["tip", "size"]).copy()
d["size"] = d["size"].astype(int)
d.shape
```

(234, 8)

234 checks from the original 247: —3 duplicates, —10 incomplete.

Put the Whole Thing in a Function

```
def load_tips(path="tips_messy.csv"):
    """Read the register export and return analysis-ready data."""
    d = pd.read_csv(path).drop(columns="server_note")
    d["total_bill"] = (d["total_bill"].str.strip()
                      .str.replace(r"[$,]", "", regex=True).astype(float))
    d["tip"] = d["tip"].replace(-99, np.nan)
    d["sex"] = d["sex"].str.strip().str.upper().str[0].map({"M": "Male", "F": "Female"})
    d["smoker"] = d["smoker"].str.strip().str.upper().str[0].map({"N": "No", "Y": "Yes"})
    d["day"] = d["day"].str.strip().str.title()
    d["time"] = d["time"].str.strip().str.title()
    d = d.drop_duplicates().dropna(subset=["tip", "size"]).copy()
    d["size"] = d["size"].astype(int)
    return d.reset_index(drop=True)
```

Why a function? It runs the same way every time, it is one line to re-run when the next month's export arrives, and it is the honest answer to "what did you do to the data?"

Validate: State Your Expectations as Code

```
tips = load_tips()

assert tips["total_bill"].between(1, 500).all(), "implausible bill"
assert tips["tip"].ge(0).all(), "negative tip"
assert tips["tip"].le(tips["total_bill"]).all(), "tip exceeds bill"
assert set(tips["sex"]) == {"Male", "Female"}, "unexpected sex"
assert set(tips["day"]) == {"Thur", "Fri", "Sat", "Sun"}, "unexpected day"
assert tips["size"].between(1, 20).all(), "implausible party"
assert not tips.isna().any().any(), "missing values remain"
print(f"all checks passed on {len(tips)} rows")
```

all checks passed on 234 rows

These assertions are the cheapest insurance in data analysis. They fail **loudly** next month when the register starts exporting something new.

Which Checks Did We Lose?

We went from 247 rows to 234. The owner will ask *which* ones — and that question is a comparison between **two columns living in two different DataFrames**.

```
lost = pd.Index(raw["check_id"]).difference(tips["check_id"])
lost.tolist()
```

```
['C0054',
 'C0061',
 'C0067',
 'C0101',
 'C0107',
 'C0129',
 'C0164',
 'C0190',
 'C0216',
 'C0233']
```

Ten checks, by name. That is R's `setdiff(raw$check_id, tips$check_id)`.

Comparing Two Columns From Two Frames

R	pandas	Returns
<code>a %in% b</code>	<code>a.isin(b)</code>	Boolean mask, aligned to a
<code>which(a %in% b)</code>	<code>np.flatnonzero(a.isin(b))</code>	positions in a
<code>a[a %in% b]</code>	<code>a[a.isin(b)]</code>	the matching values
<code>match(a, b)</code>	<code>pd.Index(b).get_indexer(a)</code>	position in b, -1 if absent
<code>intersect(a, b)</code>	<code>pd.Index(a).intersection(b)</code>	in both
<code>setdiff(a, b)</code>	<code>pd.Index(a).difference(b)</code>	in a, not b
<code>setdiff(b, a)</code>	<code>pd.Index(b).difference(a)</code>	in b, not a
<code>union(a, b)</code>	<code>pd.Index(a).union(b)</code>	either

Wrap a column in `pd.Index(...)` to get the set operations. The Index type carries them; a plain Series does not.

A Worked Example

```
a = pd.Series(["a", "b", "c"])
b = pd.Series(["b", "f", "g", "a", "i"])
```

```
[a.isin(b).tolist(),
 pd.Index(b).get_indexer(a).tolist(),
 pd.Index(a).intersection(b).tolist(),
 pd.Index(a).difference(b).tolist()]
```

```
[[True, True, False], [3, 0, -1], ['a', 'b'], ['c']]
```

`get_indexer` reads element by element: "a" sits at position 3 of b, "b" at position 0, and "c" is absent, so `-1`.

In Python `-1` is a legal position meaning “last element”, so filter on `>= 0` before using the result to index anything.

The Trap: == Is Not a Set Comparison

```
try:  
    a == b  
except ValueError as e:  
    print("ValueError:", e)
```

ValueError: Can only compare identically-labeled Series objects

Comparison between two Series aligns on the **index**, so it needs identical labels — and columns from two different frames rarely have them.

`isin`, `get_indexer`, and the Index set operations all sidestep this: they compare **values as a set**, ignoring the index and the length. That is exactly why they are the right tools for cross-DataFrame work.

Rows Lost Is Not Values Lost

```
len(raw) - len(tips), len(lost)
```

```
(13, 10)
```

Thirteen **rows** disappeared but only ten **check numbers**. The other three were the duplicate rows — and a duplicate shares its `check_id` with a row that survived, so the id never left the set.

Set operations work on values, not on rows. When the two numbers disagree, that disagreement is telling you something about duplicates. Here it confirms exactly what we expect: $13 = 10 \text{ incomplete} + 3 \text{ duplicated}$.

And Which Rows Were They?

```
raw.loc[raw["check_id"].isin(lost), ["check_id", "tip", "size", "day"]].head(5)
```

	check_id	tip	size	day
18	C0067	-99.00	2.0	Sat
25	C0129	NaN	2.0	Thur
34	C0061	NaN	2.0	Sat
65	C0233	3.39	NaN	Sat
66	C0216	-99.00	2.0	Sat

`isin` goes the other way — from the set of names back to the rows — so the two operations pair up: difference to find *which*, `isin` to pull *them*.

Before and After

```
tips.head(4)
```

	check_id	total_bill	tip	sex	smoker	day	time	size
0	C0004	23.68	3.31	Male	No	Sun	Dinner	2
1	C0083	10.07	1.83	Female	No	Thur	Lunch	1
2	C0241	27.18	2.00	Female	Yes	Sat	Dinner	2
3	C0081	19.44	3.00	Male	Yes	Thur	Lunch	2

```
tips.dtypes.to_dict()
```

```
{'check_id': dtype('O'),  
 'total_bill': dtype('float64'),  
 'tip': dtype('float64'),  
 'sex': dtype('O'),  
 'smoker': dtype('O'),  
 'day': dtype('O'),  
 'time': dtype('O'),  
 'size': dtype('int64')}
```

Section 4

Stage 3: Derive the Variables You Analyze

The Outcome Variable

The owner asked about *tips*, but the right outcome is the **tip rate**, not the dollar tip: a \$6 tip is generous on a \$25 bill and stingy on a \$80 one.

```
tips["tip_pct"] = tips["tip"] / tips["total_bill"]  
tips["tip_pct"].head(3).round(4).tolist()
```

```
[0.1398, 0.1817, 0.0736]
```

Choosing the outcome is the most consequential decision in the analysis, and it is made before any computation. Analyze `tip` and you will conclude that big parties tip well; analyze `tip_pct` and you will conclude the opposite.

Several Columns at Once with assign

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

	total_bill	size	bill_pp	is_weekend	bill_q
0	23.68	2	11.84	True	Q3
1	10.07	1	10.07	False	Q1
2	27.18	2	13.59	True	Q4
3	19.44	2	9.72	False	Q3

assign takes `lambda d: ...` so each new column can refer to the frame being built, and it returns a **new** frame rather than mutating in place.

Why Bin the Bill?

```
tips.groupby("bill_q", observed=True)["total_bill"].agg(["count", "min", "max"]).round(2)
```

	count	min	max
bill_q			
Q1	61	3.07	13.42
Q2	56	13.51	17.82
Q3	58	17.89	24.52
Q4	59	24.55	50.81

qcut makes four groups of roughly equal **count**. We will use these quartiles over and over to ask: *“is this group difference still there once I compare like with like?”*

The Analysis Frame

```
tips.shape, tips.columns.tolist()
```

```
((234, 12),  
 ['check_id',  
  'total_bill',  
  'tip',  
  'sex',  
  'smoker',  
  'day',  
  'time',  
  'size',  
  'tip_pct',  
  'bill_pp',  
  'is_weekend',  
  'bill_q'])
```

234 checks, 11 columns: 8 as recorded, 3 derived. This is the object every remaining stage works from — and the function that builds it fits on one slide.

Section 5

Stage 4: Look at One Variable

How Does the Tip Rate Behave?

```
tips["tip_pct"].describe().round(4)
```

count	234.000
mean	0.160
std	0.061
min	0.036
25%	0.127
50%	0.154
75%	0.191
max	0.710

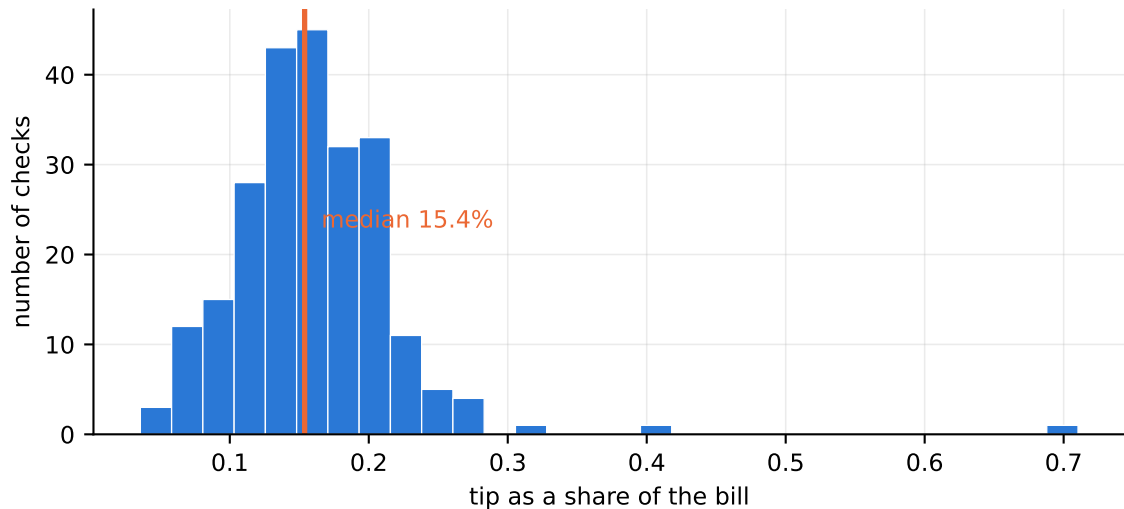
Name: tip_pct, dtype: float64

Median 15.4%, mean 16.0%. **Mean above median** means a right skew — a few very large tip rates pulling the average up. The maximum of 71% confirms it.

A Picture Beats the Six Numbers

```
fig, ax = plt.subplots()
ax.hist(tips["tip_pct"], bins=30, color=BLUE,
        edgecolor="white", linewidth=0.5)
ax.axvline(tips["tip_pct"].median(), color=ORANGE, lw=2)
ax.set_xlabel("tip as a share of the bill")
ax.set_ylabel("number of checks")
plt.show()
```

One Tall Cluster, and a Long Thin Tail



Does the 15–20% Norm Hold?

```
norm = tips["tip_pct"].between(0.15, 0.20)
print(f"within 15-20%: {norm.mean():.1%}")
print(f"below 15%      : {(tips['tip_pct'] < 0.15).mean():.1%}")
print(f"above 20%      : {(tips['tip_pct'] > 0.20).mean():.1%}")
```

```
within 15-20%: 38.9%
below 15%     : 45.3%
above 20%     : 15.8%
```

Finding 1. The median check tips 15.4%, right at the bottom of the norm — but the norm describes a minority. Only 39% of parties land inside it, and **45% tip less than 15%**. “Most people tip 15 to 20 percent” is false here.

The Right Tail

```
tips.nlargest(4, "tip_pct")[["check_id", "total_bill", "tip", "tip_pct", "size"]]
```

	check_id	total_bill	tip	tip_pct	size
211	C0173	7.25	5.15	0.710	2
57	C0179	9.60	4.00	0.417	2
230	C0068	3.07	1.00	0.326	1
199	C0184	23.17	6.50	0.281	4

All four are **small bills**. A \$5.15 tip on a \$7.25 bill is 71%, but in dollars it is unremarkable. Extreme *rates* live where the denominator is small.

The Left Tail

```
tips.nsmallest(3, "tip_pct")[["check_id", "total_bill", "tip", "tip_pct", "size"]]
```

	check_id	total_bill	tip	tip_pct	size
141	C0238	32.83	1.17	0.036	2
106	C0103	44.30	2.50	0.056	3
23	C0058	26.41	1.50	0.057	2

The mirror image: the stingiest *rates* are on the **largest** bills.

Two tails, one pattern — and it is a hint about Stage 5.

Do We Drop the Outliers?

No. And it is worth being explicit about why.

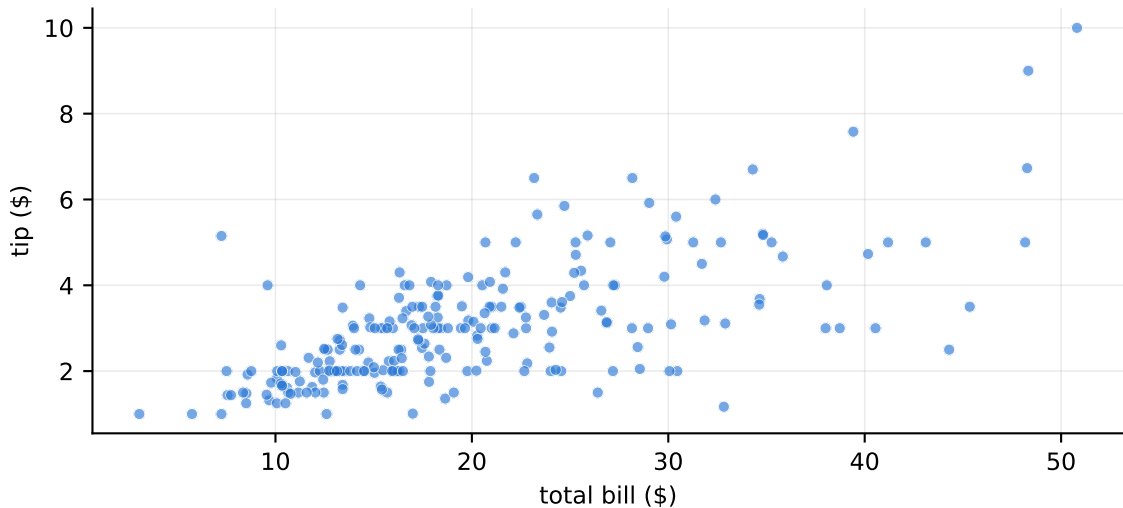
- They are **not errors**. A 71% tip on a \$7.25 check is unusual, not impossible, and our validation confirmed $\text{tip} \leq \text{bill}$.
- They are **not noise** — they are the clearest cases of the very pattern we are about to find.
- Dropping points because they are inconvenient is how analyses become dishonest.

What we do instead: report the **median** alongside the mean, since the median is barely moved by them, and keep every row.

Section 6

Stage 5: Two Variables

Does the Tip Follow the Bill?



Putting a Number on It

```
tips[["total_bill", "tip", "size", "tip_pct"]].corr().round(3)
```

	total_bill	tip	size	tip_pct
total_bill	1.000	0.674	0.594	-0.340
tip	0.674	1.000	0.488	0.341
size	0.594	0.488	1.000	-0.141
tip_pct	-0.340	0.341	-0.141	1.000

tip and total_bill correlate at 0.68 — strong, as expected.

But look at the last column: **tip_pct correlates with total_bill at -0.34** . Bigger bills get bigger tips *and* smaller tip rates. Both at once.

A Line Through the Cloud

```
b1, b0 = np.polyfit(tips["total_bill"], tips["tip"], 1)
print(f"tip = {b0:.2f} + {b1:.3f} x bill")
```

$\text{tip} = 0.92 + 0.105 \times \text{bill}$

Read it out loud: **a tip starts at about 92 cents and adds about 10.5 cents per dollar of bill.**

That positive intercept is the whole story. If tipping were a pure percentage, the line would pass through the origin.

The Mechanism

If $\text{tip} = 0.92 + 0.105 \times \text{bill}$, then

$$\text{tip}\% = \frac{0.92}{\text{bill}} + 0.105$$

The fixed 92 cents is a **shrinking share** of a growing bill:

Bill	\$0.92 is	plus	predicted tip %
\$10	9.2%	10.5%	19.7%
\$20	4.6%	10.5%	15.1%
\$40	2.3%	10.5%	12.8%

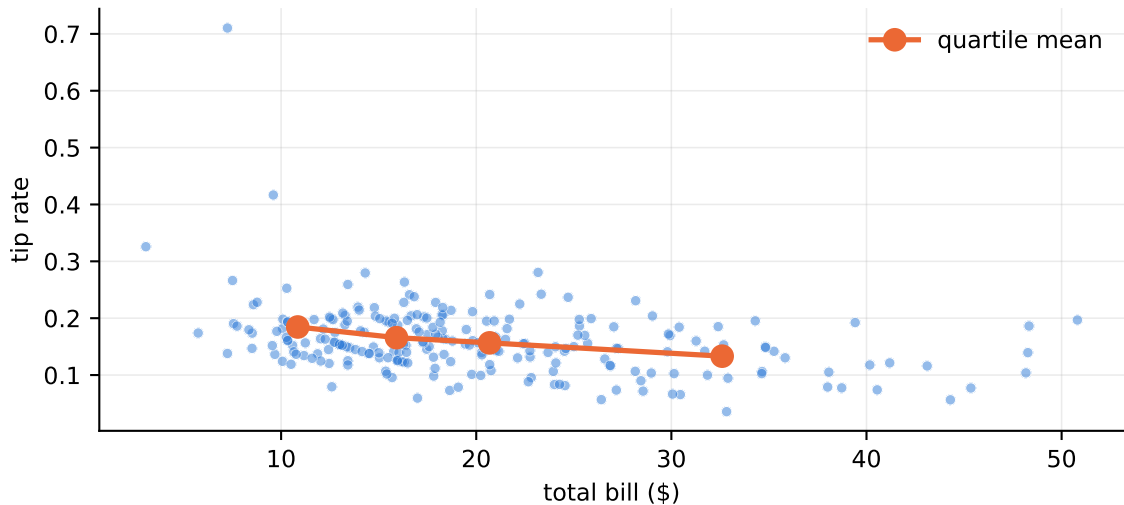
The falling tip rate is not a change in generosity. It is arithmetic.

Seeing It Directly

```
cent = tips.groupby("bill_q", observed=True)["total_bill"].mean()
qm    = tips.groupby("bill_q", observed=True)["tip_pct"].mean()

ax.scatter(tips["total_bill"], tips["tip_pct"], s=16, color=BLUE, alpha=0.5)
ax.plot(cent, qm, color=ORANGE, lw=2, marker="o", label="quartile mean")
```

The Tip Rate Against the Bill: The Quartile Means Slope Down



The Quartile Table

```
tips.groupby("bill_q", observed=True)["tip_pct"].agg(["count", "mean", "median"]).round(4)
```

	count	mean	median
bill_q			
Q1	61	0.184	0.167
Q2	56	0.166	0.155
Q3	58	0.157	0.155
Q4	59	0.133	0.139

Finding 2. The tip rate falls steadily from 18.5% on the smallest quarter of bills to 13.3% on the largest — a drop of more than five percentage points. The median moves the same way, so this is not driven by the outliers.

Spread Grows Too

```
tips.groupby("bill_q", observed=True)["tip"].agg(["mean", "std"]).round(2)
```

	mean	std
bill_q		
Q1	1.93	0.70
Q2	2.63	0.72
Q3	3.22	0.95
Q4	4.29	1.70

The standard deviation of the dollar tip more than doubles from Q1 to Q4. Large bills are not just bigger — they are **less predictable**.

(A statistician calls this *heteroscedasticity*; it is the reason the scatter fans out to the right, and it matters if you later fit a regression.)

What That Means for the Restaurant

Finding 2, stated for the owner:

*A server who turns a \$20 table is likely to see about 15% of it. A server who turns a \$40 table sees about 13% — but that is \$5.20 instead of \$3.00. **The rate falls; the money rises.***

This is why the outcome variable mattered so much. “Big parties tip badly” and “big parties are worth more” are both true, and they are answers to different questions.

Section 7

Stage 6: Compare Groups — Carefully

The First Cut

```
tips.groupby("smoker")["tip_pct"].agg(["count", "mean", "median"]).round(4)
```

	count	mean	median
smoker			
No	146	0.159	0.156
Yes	88	0.163	0.154

Smokers 16.3%, non-smokers 15.9%. A gap of 0.4 percentage points.

If you stopped here you would report “no real difference between smokers and non-smokers.”

That conclusion would be wrong — not because the means are wrong, but because the mean is the wrong summary.

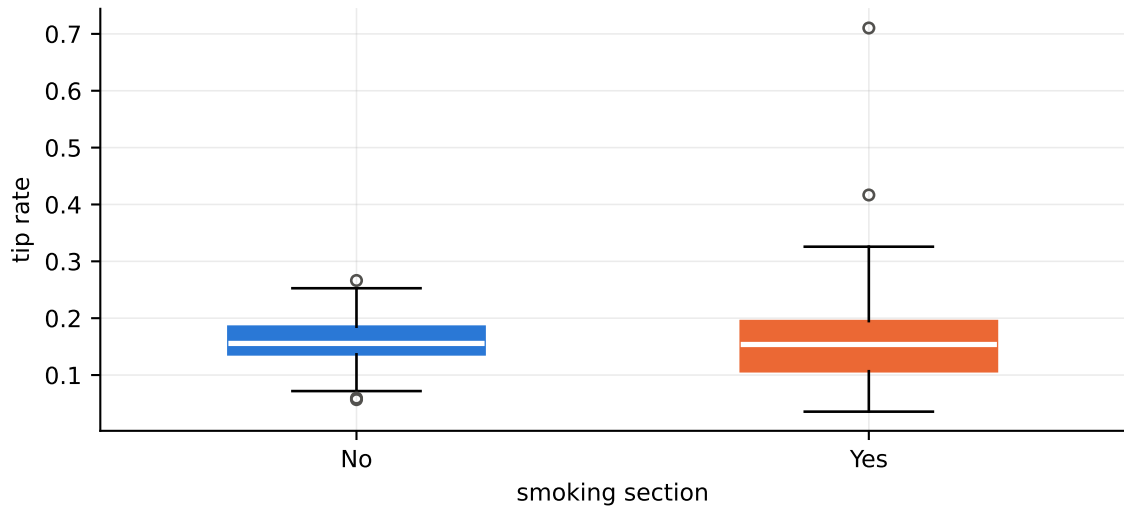
Look at the Spread

```
tips.groupby("smoker")["tip_pct"].agg(["mean", "std", "min", "max"]).round(4)
```

	mean	std	min	max
smoker				
No	0.159	0.039	0.057	0.266
Yes	0.163	0.087	0.036	0.710

The standard deviation among smokers is **more than twice** that of non-smokers. Same center, utterly different behavior.

Boxes Level, Whiskers Far Longer



Quantifying the Tails

```
tips.groupby("smoker")["tip_pct"].apply(  
    lambda s: pd.Series({"under 10%": (s < 0.10).mean(),  
                          "over 25%": (s > 0.25).mean()})).unstack().round(3)
```

	under 10%	over 25%
smoker		
No	0.062	0.014
Yes	0.193	0.080

Finding 3. Smoking-section parties are **three times** as likely to tip under 10% (19.3% vs 6.2%) and nearly six times as likely to tip over 25%. The averages match; the behavior does not.

Lesson: compare distributions, not just centers. Two groups with identical means can be completely different.

A Raw Difference by Sex

```
tips.groupby("sex")[["total_bill", "tip_pct"]].mean().round(4)
```

	total_bill	tip_pct
sex		
Female	18.300	0.167
Male	20.911	0.157

Women tip 16.7%, men 15.7% — a full percentage point. A tempting headline.

But look at the **first** column before you write it.

The Confound

Women in this sample pay **smaller bills** (\$18.30 vs \$20.91).

And we already know from Stage 5 that smaller bills carry **higher tip rates**, for reasons that have nothing to do with who is paying.

$\text{sex} \longrightarrow \text{tip_pct} \quad ?$

$\text{sex} \longrightarrow \text{total_bill} \longrightarrow \text{tip_pct} \quad \text{or this?}$

The fix: compare men and women *at the same bill size*. That is what stratification means, and `pivot_table` does it in one line.

Stratifying

```
tips.pivot_table(index="bill_q", columns="sex", values="tip_pct",  
                  aggfunc="mean", observed=True).round(4)
```

sex	Female	Male
bill_q		
Q1	0.187	0.182
Q2	0.175	0.159
Q3	0.152	0.158
Q4	0.137	0.131

Finding 4. Within every bill quartile the gap collapses to a fraction of a point, and it **changes direction** twice. The apparent sex difference was a bill-size difference wearing a disguise.

The General Lesson

A difference between two groups is only evidence about those groups if everything else is comparable.

In this dataset, *every* group differs in average bill size — so every raw group comparison is partly a restatement of Finding 2.

Before you believe a group difference	pandas
Are the groups the same size?	<code>groupby(g).size()</code>
Do they differ on the main driver?	<code>groupby(g)[driver].mean()</code>
Does the gap survive stratification?	<code>pivot_table(index=bin, columns=g)</code>
Is it spread rather than center?	<code>groupby(g)[y].std()</code>

Day and Time: Check the Cross-Tab First

```
pd.crosstab(tips["day"], tips["time"])
```

time	Dinner	Lunch
Fri	11	7
Sat	82	0
Sun	73	0
Thur	1	60

Stop. Saturday and Sunday are dinner-only. Thursday is lunch except for a single check. There is almost no overlap.

Why That Matters

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

```
time
Dinner    0.159
Lunch     0.164
Name: tip_pct, dtype: float64
```

“Lunch tips better than dinner” and “Thursday tips better than Saturday” are **the same sentence** in this dataset. The design does not let you separate the day from the meal.

No amount of cleverness fixes this — it is a property of the data, not the method. The honest move is to say so.

This is why crosstab belongs *before* groupby, not after.

What We Can Still Say About Days

```
tips.groupby("day").agg(n=("tip", "size"),  
                        bill=("total_bill", "mean"),  
                        pct=("tip_pct", "mean")).round(3)
```

	n	bill	pct
day			
Fri	18	17.474	0.167
Sat	82	20.691	0.152
Sun	73	21.648	0.167
Thur	61	17.786	0.161

Saturday is lowest at 15.2% — and Saturday also has the second-highest average bill. With $n = 18$ on Friday, the day-to-day differences are small relative to the noise.

One Table for the Owner

```
tips.pivot_table(index="time", columns="smoker", values="tip_pct",  
                  aggfunc=["mean", "count"], margins=True).round(3)
```

	mean			count		
	No	Yes	All	No	Yes	All
time						
Dinner	0.158	0.160	0.159	102	65	167
Lunch	0.161	0.170	0.164	44	23	67
All	0.158	0.163	0.160	146	88	234

transform: Comparing Like With Like

```
tips["pct_vs_peers"] = (tips["tip_pct"]  
                        - tips.groupby("bill_q", observed=True)["tip_pct"].transform("mean"))  
tips.groupby("sex")["pct_vs_peers"].mean().round(4)
```

```
sex  
Female    0.004  
Male      -0.002  
Name: pct_vs_peers, dtype: float64
```

transform broadcasts each quartile's mean back onto every row, so subtracting it removes the bill-size effect from every check at once.

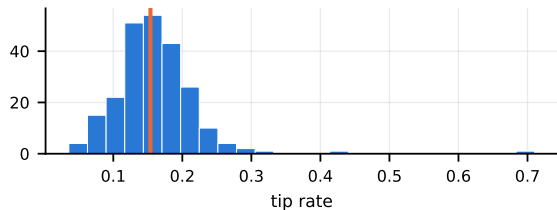
The residual sex gap is **under a quarter of a percentage point** — the same conclusion as the stratified table, in one line instead of a grid.

Section 8

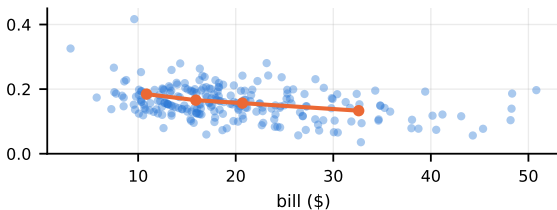
Stage 7: One Figure That Tells the Story

The Summary Panel

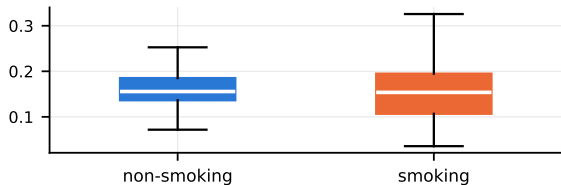
1. Median tip 15.4%, long right tail



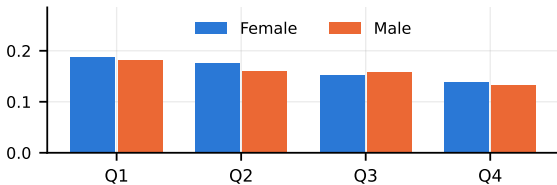
2. Rate falls as the bill grows



3. Same centre, double the spread



4. No sex gap within bill quartiles



Panel 3 hides the outlying points so the boxes stay readable; they are shown in full on the boxplot slide.

Building That Panel

```
fig, axes = plt.subplots(2, 2, figsize=(8.2, 3.9))

axes[0, 0].hist(tips["tip_pct"], bins=25, color=BLUE)
axes[0, 1].scatter(tips["total_bill"], tips["tip_pct"], s=9, color=BLUE)
axes[1, 0].boxplot([...], patch_artist=True)
axes[1, 1].bar(xs - .19, piv["Female"], .38, color=BLUE, label="Female")

fig.tight_layout()
fig.savefig("tips_summary.png", dpi=200, bbox_inches="tight")
```

Figure rules that matter: one message per panel, the message in the panel title, a legend whenever there are two series, and never color alone to carry meaning.

Section 9

Stage 8: Report

The Findings

- ❶ **The norm is a myth here.** The median check tips 15.4%, but only 39% of parties fall in the 15–20% band and 45% tip less than 15%.
- ❷ **The tip rate falls with the bill,** from 18.5% on the smallest quarter of checks to 13.3% on the largest. The cause is arithmetic: tips behave like $\$0.92 + 10.5\%$ of the bill, and the fixed part shrinks as a share.
- ❸ **Smoking-section parties are not worse tippers — they are more variable ones.** The means differ by 0.4 points, but they are three times as likely to tip under 10%.
- ❹ **There is no sex difference.** The raw one-point gap disappears entirely once men and women are compared at the same bill size.

What We Cannot Say

An analysis is judged as much by its stated limits as by its findings.

- **Day and meal cannot be separated.** Saturday and Sunday are dinner-only, Thursday is lunch-only. Any “weekend effect” here is also a “dinner effect.”
- **One restaurant, one server, about ten weeks.** Nothing here generalizes to restaurants in general.
- **234 of 247 checks.** We dropped 10 incomplete records and 3 duplicates; the dropped rows were spread across days, but we cannot prove they were missing at random.
- **Nothing here is causal.** We observed what happened; we changed nothing.
- **Small cells.** Friday has 18 checks. Treat its average as a hint, not a number.

The Whole Analysis

```
tips = (load_tips("tips_messy.csv")
        .assign(tip_pct = lambda d: d["tip"] / d["total_bill"],
                bill_pp = lambda d: d["total_bill"] / d["size"],
                bill_q   = lambda d: pd.qcut(d["total_bill"], 4,
                                            labels=["Q1", "Q2", "Q3", "Q4"])))

headline = tips.groupby("bill_q", observed=True)["tip_pct"].mean()
by_smoker = tips.groupby("smoker")["tip_pct"].agg(["mean", "std"])
by_sex = tips.pivot_table(index="bill_q", columns="sex", values="tip_pct")
```

Twelve lines, from a broken CSV to every number in the report. That is the goal: an analysis someone else can re-run and get your answer.

A Checklist You Can Reuse

Stage	Do not skip
Load	<code>shape</code> , <code>dtypes</code> , <code>isna().sum()</code> , <code>duplicated().sum()</code> <code>value_counts()</code> on every category
Clean	work on a copy; write the defect inventory first count NaNs after every <code>map</code> and every type conversion assert your expectations; check who you are dropping
Derive	choose the outcome deliberately; bin the main driver
Explore	median beside mean; look before you summarize
Compare	<code>crosstab</code> before <code>groupby</code> ; stratify every raw gap spread, not just centre
Report	state the limits as plainly as the findings

References

- Bryant, P.G. & Smith, M.A. (1995) *Practical Data Analysis: Case Studies in Business Statistics*. Irwin — the original tips data.
- McKinney, W. *Python for Data Analysis*, 3rd ed.
 - Data cleaning and preparation: <https://wesmckinney.com/book/data-cleaning>
 - Data aggregation and group operations: <https://wesmckinney.com/book/data-aggregation>
 - Plotting and visualization: <https://wesmckinney.com/book/plotting-and-visualization>
- Wickham, H. (2014) “Tidy Data”, *Journal of Statistical Software* 59(10).
<https://doi.org/10.18637/jss.v059.i10>
- Irizarry, R.A. *Introduction to Data Science* — on confounding and stratification.
- pandas user guide: https://pandas.pydata.org/docs/user_guide/