Data 201

Intro to Data Analysis in Python

Data Wrangling with pandas for R Users

Week 2

Dr. Rebin Muhammad

Welcome – Week 2

Last week: Python fundamentals, NumPy, and the R→Python mindset. This week we focus on **tabular data**: the pandas DataFrame and how it maps from dplyr.

Today : DataFrame basics, dplyr→pandas verbs, method chaining, grouped summaries, missing data, and hands-on translation.

Have the **Week 2 Notebook** and your **R→Python cheat sheet** open.

What we'll do today (Week 2)

1. **pandas in the ecosystem** — Where it fits (NumPy → pandas → seaborn/statsmodels)
2. **DataFrame basics** — Create, inspect, columns and rows
3. **dplyr → pandas** — select, filter, mutate, arrange (with correct boolean logic)
4. **Pipes → method chaining** — Build pipelines step by step
5. **group_by and summarize** — Grouped summaries with `.groupby()` and `.agg()`
6. **Missing data** — Detect, drop, fill
7. **Practice** — Translate a dplyr pipeline into pandas

Learning goals

By the end of today you will be able to:

1. **Explain** how pandas DataFrames relate to R data frames
2. **Perform** filtering, selecting, mutating, and sorting in pandas
3. **Use method chaining** instead of R pipes (`%>%`)
4. **Create grouped summaries** with `.groupby()` and `.agg()`

pandas in the Python ecosystem

Where pandas fits:

- **NumPy** — numerical foundation (arrays, vectorization)
- **pandas** — tabular data: columns, rows, and dplyr-like operations
- **seaborn / statsmodels** — build on pandas DataFrames

Think of pandas as: R data frames + dplyr verbs. Same mental model; different syntax.

DataFrame basics: R vs pandas

Task	R	pandas
Create	`data.frame(...)`	`pd.DataFrame(...)`
Structure / types	`str(df)`	`df.info()`
First rows	`head(df)`	`df.head()`
Dimensions	`dim(df)`	`df.shape`
Column names	`names(df)`	`df.columns`

Same idea: a rectangular table with named columns and (usually) a row index.

➡ Notebook Section 1

Columns and rows

Accessing columns: one column → Series; multiple → DataFrame.

```
df["price"]           # one column → Series
df[["price", "size"]] # multiple columns → DataFrame
```

Filtering rows: pass a boolean Series (same length as the DataFrame).

```
df[df["price"] > 300]
```

➡ Notebook Section 2

Boolean logic (critical for filters)

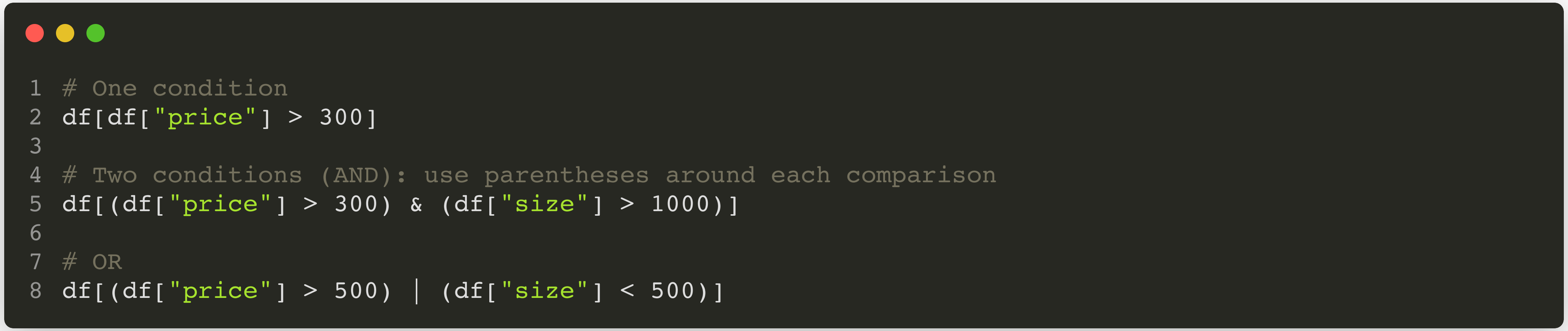
In R you use `&` and |; pandas is the same, but **you must use parentheses** so each condition is a complete expression.

R	**pandas**
`&` and |	`&` and |
Vector recycling	No recycling (aligns by index)
Parentheses often optional	**Parentheses required**

Wrong: `df[df["price"] > 300 & df["size"] > 1000]` → operator precedence error.

Right: `df[(df["price"] > 300) & (df["size"] > 1000)]`

Boolean logic – example



```python
# One condition
df[df["price"] > 300]

# Two conditions (AND): use parentheses around each comparison
df[(df["price"] > 300) & (df["size"] > 1000)]

# OR
df[(df["price"] > 500) | (df["size"] < 500)]
```

Rule of thumb: wrap each comparison in parentheses when combining with `&` or `|`.

select() → column subsetting

dplyr	pandas
`select(a, b)`	`df[["a", "b"]]`

- **No tidyselect helpers by default** — use explicit column names (or a list of names).
- Single column: `df["a"]` (Series). Multiple: `df[["a", "b"]]` (DataFrame).

```python
df[["price", "size", "neighborhood"]]
```

➡ Cheat Sheet Section 6

filter() → row filtering

Option 1: Boolean indexing (like R's logical subsetting).

```python
df[df["price"] > 200]
```

Option 2: `.query()` — often feels closest to dplyr.

```python
df.query("price > 200")
df.query("price > 200 and size > 1000")
```

Note: `.query()` uses string expressions; column names must be valid Python identifiers (or use backticks in the string for special names).

➡ Notebook Section 2

mutate() → .assign()

dplyr	pandas
`mutate(ppsqft = price / size)`	`.assign(ppsqft=lambda d: d.price / d.size)`

Why lambda? The argument is the (current) DataFrame. Using `d` avoids confusion with partially-created columns and makes each new column depend only on existing columns.

```
df.assign(ppsqft=lambda d: d.price / d.size)
```

Important: `.assign()` returns a *new* DataFrame; it does not modify the original (immutable style).

➡ Notebook Section 3

arrange() → .sort_values()

dplyr	pandas
`arrange(price)`	`.sort_values("price")`
`arrange(desc(price))`	`.sort_values("price", ascending=False)`
Multiple columns	`.sort_values(["col1", "col2"])`

```python
df.sort_values("price")
df.sort_values("price", ascending=False)
df.sort_values(["neighborhood", "price"])
```

Pipes → method chaining

R: `%>%` passes the result of the left side as the first argument to the right.

Python: Each method returns a DataFrame, so you chain with dots. Use parentheses so you can break the chain across lines.

R:

```r
df %>%
  filter(price > 200) %>%
  mutate(ppsqft = price / size)
```

Python:

```python
(df
 .query("price > 200")
 .assign(ppsqft=lambda d: d.price / d.size))
```

Why method chaining helps

- **Readable:** Top to bottom = order of operations (filter → mutate → ...).
- **No temporary variables:** Each step is the input to the next.
- **Same as dplyr:** If you're used to pipes, chaining will feel familiar.

Style: Put the opening `(` on the same line as the object, then each method on its own line with a leading dot.

group_by() → .groupby()

dplyr	pandas
`group_by(x)`	`.groupby("x")`
`summarize(...)`	`.agg(...)`

Typical pattern: `.groupby("col").agg(new_name=("existing_col", "function"))`

```python
df.groupby("neighborhood").agg(mean_price=("price", "mean"))
df.groupby("type").agg(
    mean_price=("price", "mean"),
    count=("price", "count")
)
```

➡ Notebook Section 4

Grouped summary – full example

R:

```r
df %>%
  group_by(neighborhood) %>%
  summarize(mean_price = mean(price), n = n())
```

Python:

```python
df.groupby("neighborhood").agg(
    mean_price=("price", "mean"),
    n=("price", "count")
)
```

Note: pandas uses "count" or "size" for row counts; here `("price", "count")` counts non-null `price` values.

Missing data in pandas

Task	**R**	**pandas**
Detect	`is.na(x)`	`df.isna()` or `pd.isna()`
Drop	`na.omit()`	`.dropna()`
Fill	`replace()`, `ifelse()`	`.fillna()`

```python
df.isna()                       # DataFrame of True/False
df.dropna()                     # drop rows with any NA
df.dropna(subset=["price"])  # drop only where price is NA
df.fillna(0)                    # fill NAs with 0
```

➡ Cheat Sheet Section 7

Series vs DataFrame (a common pitfall)

- **One column** `df["price"]` → **Series** (1D). Result of `.agg("mean")` on a single column can be a Series.
- **Two or more columns** `df[["price", "size"]]` → **DataFrame** (2D).

Why it matters: Some operations expect a DataFrame (e.g. chaining `.assign()`). If you have a Series, you may need to wrap or use `.to_frame()`.

Rule of thumb: Subset columns with `df[["a", "b"]]` when you want to keep a DataFrame.

Common pandas pitfalls

- **Forgetting parentheses in filters** — `(df["price"] > 300) & (df["size"] > 1000)` needs parentheses around each comparison.
- **Confusing Series vs DataFrame** — Single column is a Series; use `df[["col"]]` for a one-column DataFrame.
- **Modifying views vs copies** — Prefer `.assign()` and new objects; avoid chained indexing (e.g. `df[df.x > 0]["y"] = 1`).
- **Overusing loops** — Use vectorized operations and `.apply()` when appropriate; avoid row-by-row loops on large data.

➡ Notebook Section 5

Active learning (Part 1) — 20–25 min

Translate this dplyr pipeline to pandas.

R code:

```r
df %>%
  filter(price > 300, size > 1000) %>%
  mutate(ppsqft = price / size) %>%
  group_by(type) %>%
  summarize(mean_ppsqft = mean(ppsqft))
```

Tasks:

1. **Predict** — What is the structure of the result? (One row per what? Which columns?)
2. **Translate** — Write the pandas version (`.query()` or boolean filter, `.assign()`, `.groupby()`, `.agg()`).
3. **Run** — Execute in the Week 2 Notebook and interpret.

➡ Notebook Section 6

Active learning – solution sketch

```python
(df
 .query("price > 300 and size > 1000")
 .assign(ppsqft=lambda d: d.price / d.size)
 .groupby("type")
 .agg(mean_ppsqft=("ppsqft", "mean"))
)
```

Result: One row per `type`; column `mean_ppsqft` = mean of `price/size` in each group.

Compare with your solution. Questions on `.query()`, `.assign()`, or `.agg()`?

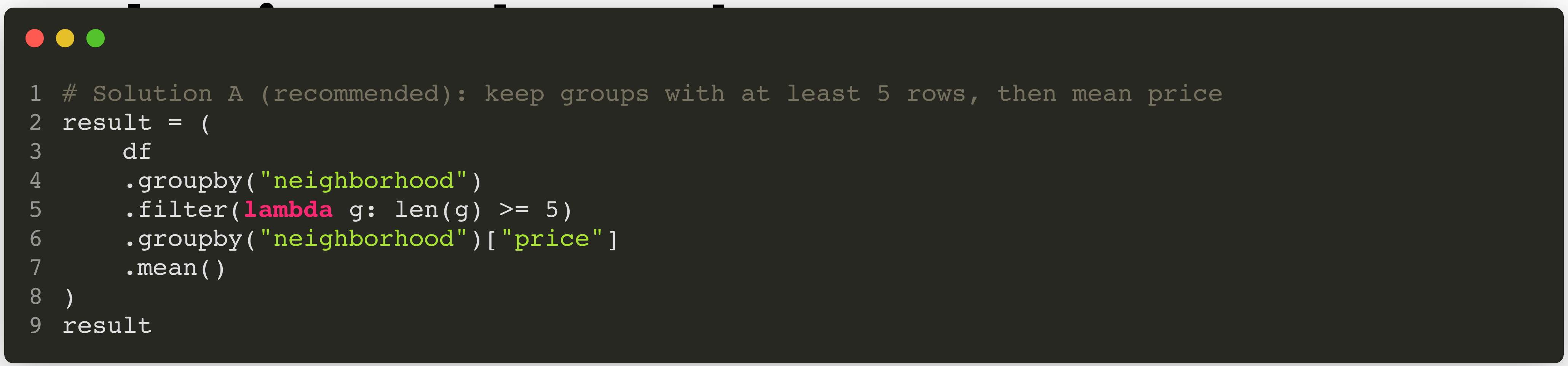
Active learning (Part 2) — if time, 10–15 min

Extend the pipeline: From the same `df`, add a step that keeps only neighborhoods with at least 5 rows, then compute mean price by neighborhood.

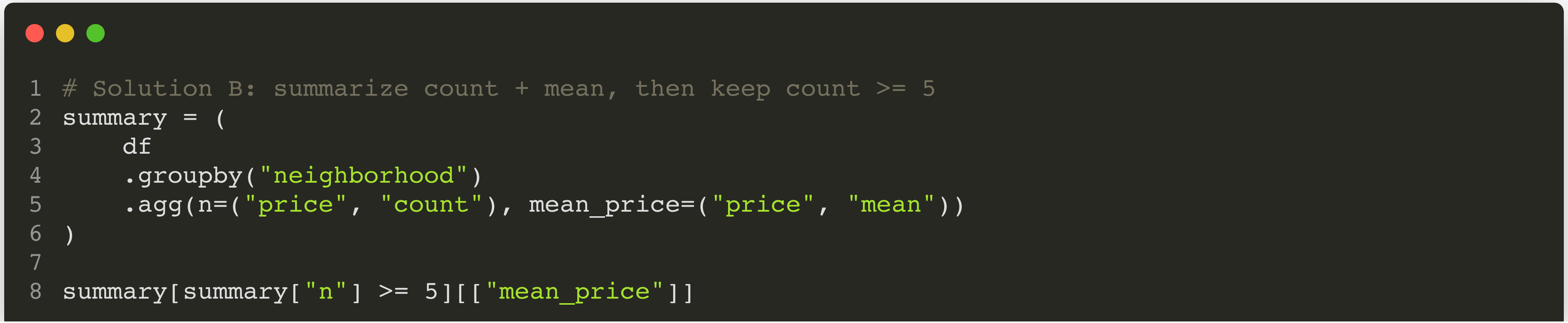
Hints: `.groupby().filter()` or `.groupby().agg()` then filter the result; or use `.groupby().agg()` with a count column and then subset rows.

➡ Use the Week 2 Notebook; compare with a neighbor or the solution.

Active learning (Part 2) –



```python
# Solution A (recommended): keep groups with at least 5 rows, then mean price
result = (
    df
    .groupby("neighborhood")
    .filter(lambda g: len(g) >= 5)
    .groupby("neighborhood")["price"]
    .mean()
)
result
```



```python
# Solution B: summarize count + mean, then keep count >= 5
summary = (
    df
    .groupby("neighborhood")
    .agg(n=("price", "count"), mean_price=("price", "mean"))
)

summary[summary["n"] >= 5][["mean_price"]]
```

Key takeaways

1. **pandas mirrors dplyr conceptually** — select, filter, mutate, arrange, group_by, summarize all have direct equivalents.
2. **Syntax is more explicit** — Column names in quotes; boolean filters need parentheses; new columns via `.assign(..., lambda d: ...)`.
3. **Method chaining replaces pipes** — `(df .query(...) .assign(...) .groupby(...) .agg(...))` reads like a dplyr pipeline.
4. **Series vs DataFrame** — Know when you have one column (Series) vs a table (DataFrame); use `df[["col"]]` when you need a DataFrame.

Looking ahead

Next week (Week 2-3): Visualization in Python

- ggplot2 → seaborn
- Grammar-of-graphics thinking (data, aesthetics, geoms) mostly survives
- Bring your R→Python cheat sheet

Week 2’s wrangling skills are what you’ll feed into plots and models in the rest of the course.

Thank you

Data 201 · Intro to Data Analysis in Python

Week 2-Part 1 — Data Wrangling with pandas for R Users