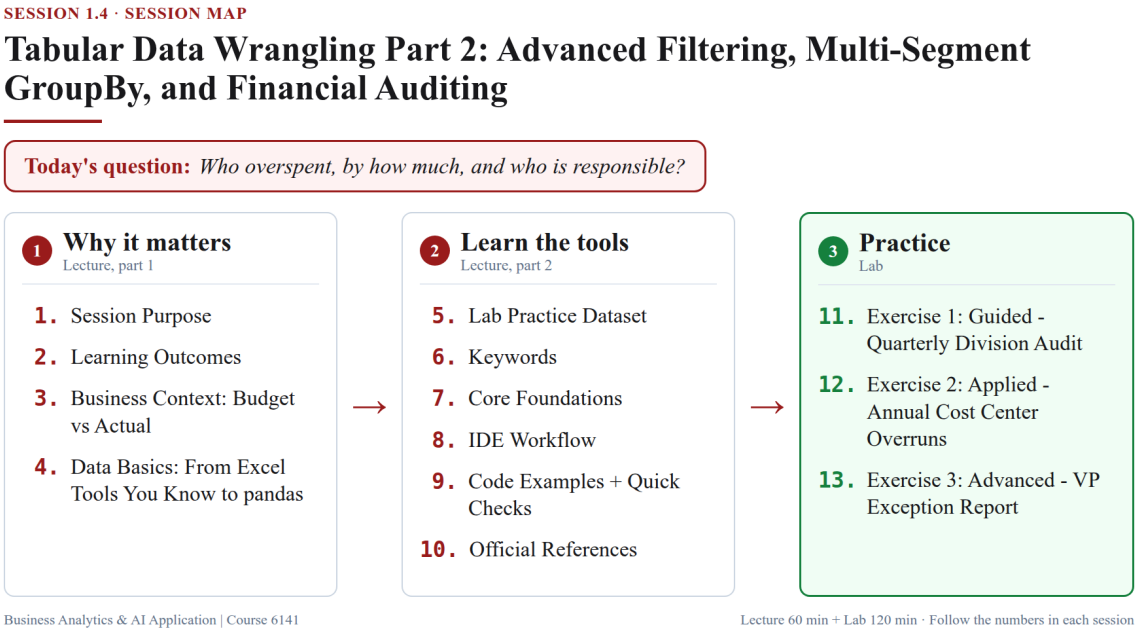
Session 1.4: Tabular Data Wrangling Part 2: Advanced Filtering, Multi-Segment GroupBy, and Financial Auditing

Course 6141: Business Analytics & AI Application | Reading for Week 4



Schedule: Week 4 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- Each division gets a spending budget every quarter, and finance compares it with what was actually spent
- Our case: 4 divisions x 4 quarters = 16 rows in one CSV file
- The finance director asks: who overspent, by how much, and who is responsible? Today, Python answers this for the whole year at once

2. Expected Learning Outcomes (By the end of this session, you will be able to)

1. "Which quarters went over budget?": filter rows with a Boolean mask (a `True`/`False` value for every row) and &

2. "How much did each division overspend this year?": summarize groups with `groupby()` (split rows into groups, like an Excel pivot table, which is a summary table with one row per group) and `.agg()`
3. "Who is responsible?": join tables with `pd.merge()` (join two tables on a shared column, like Excel's `VLOOKUP` formula, which looks up a value in another table) and fix a `KeyError`

3. Business Context: Budget vs Actual

- Four terms, one example (Consumer division, Q1):

Term	Meaning	Example
Budget	planned spending	`780,000`
Actual	real spending	`810,000`
Variance	Actual - Budget	`+30,000`
Variance %	Variance / Budget x 100	`+3.85%`

- Positive variance = overspent; negative variance = under budget.
- Cost center: the accounting code a division's spending is charged to, such as `CC-202` for Consumer. Each cost center has a responsible VP (Vice President).
- Company rule in our data: a quarter is flagged (`Audit_Status` = `Variance_Flag`) when the variance is more than 2%.
- Consumer division by quarter: Q1 `+3.85%` (flagged), Q2 `+1.92%` (over, within the rule), Q3 `+5.77%` (flagged), Q4 `-0.64%` (under budget).

4. Data Basics: From Excel Tools You Know to pandas

- Three Excel tools, in one line each:
  - **Filter:** hides the rows that do not match a rule, such as showing only the Consumer division.
  - **Pivot Table:** summarizes a large table by group. For example, put `Division_Name` in Rows and `Actual_Expense_USD` in Values to get total spending per division: 4 rows instead of 16.
  - **`VLOOKUP`:** a formula that looks up a value, such as a cost center code, in another table and returns a value from the same row, such as the VP's name. You tell it which column to return by number (for example, the 2nd column).
- The same three jobs in pandas:
  - Filter -> Boolean mask `df[mask]`: the rule is written in code, so it repeats exactly next quarter.
  - Pivot Table -> `df.groupby().agg()`: one line; run it again when new data arrives.
  - `VLOOKUP` -> `pd.merge()`: `VLOOKUP` returns a column by number; `pd.merge()` matches rows by column name, such as `Cost_Center_Code`.

5. Lab Practice Dataset

- File Link: `data/session_01_04_dataset.csv`
- Audited 16-row quarterly budget dataset (4 divisions x 4 quarters; each row is one division's spending in one quarter):
  - `Transaction_ID`: Unique record identifier (e.g., `TXN-5001`).
  - `Division_Name`: Operating business unit (Cloud, Consumer, Enterprise, Logistics).
  - `Cost_Center_Code`: Accounting cost center (CC-101, CC-202, CC-303, CC-404).
  - `Fiscal_Quarter`: Financial reporting quarter (Q1, Q2, Q3, Q4).
  - `Budget_Expense_USD`: Authorized spending budget in U.S. Dollars.
  - `Actual_Expense_USD`: Realized expenditure in U.S. Dollars.
  - `Audit_Status`: Internal audit classification (`Verified`, or `Variance_Flag` when spending is more than 2% over budget).

6. Keywords (Core Programming Concepts)

- `Boolean Mask`:
  - What: one `True` or `False` per row
  - Why: picks exactly the rows that meet a rule
- `& | ~`:
  - What: "and", "or", "not" for masks
  - Why: Python's `and` / `or` fail on whole columns
- `Split-Apply-Combine`:
  - What: split rows into groups, calculate each group, combine the results
  - Why: the idea behind every pivot table
- `groupby()` + `.agg()`:
  - What: `groupby()` splits; `.agg()` sums, counts, or averages each group
  - Why: yearly totals per division in one step
- `pd.merge()`:
  - What: joins two tables row by row
  - Why: adds facts from another table, such as the VP's name, with no copy-paste
- `Key Column`:
  - What: a column both tables share, such as `Cost_Center_Code`
  - Why: tells pandas which rows belong together

7. Core Programming & Computational Foundations

- **7.1 Combining rules: `&`, not `and`**

– *The idea:* A rule on a column, such as `df["Division_Name"] == "Consumer"`, does not give one answer. It gives one `True` or `False` **for every row**: a whole column of answers, called a Boolean mask. To combine two rules, Python must compare the two columns row by row: row 1 with row 1, row 2 with row 2, and so on. pandas uses `&` ("and"), `|` ("or"), and `~` ("not") for this row-by-row work.

– *See it on real data:* the first 4 rows of our dataset.

Row	Division	`is_consumer`	`is_over`	Both true (&)	Either true (or)
`TXN-5000`	Cloud	`False`	`False`	`False`	`False`
`TXN-5001`	Consumer	`True`	`True`	`True`	`True`
`TXN-5002`	Enterprise	`False`	`False`	`False`	`False`
`TXN-5003`	Logistics	`False`	`True`	`False`	`True`

Here `is_consumer = df["Division_Name"] == "Consumer"` and `is_over = df["Actual_Expense_USD"] > df["Budget_Expense_USD"]`. With `is_consumer & is_over`, a row is kept only when **both** columns say `True` (`TXN-5001`). With `is_consumer | is_over`, one `True` is enough (`TXN-5001` and `TXN-5003`).

– *Common mistake 1, using `and`:* Python's own `and` can only judge one `True` or `False` at a time. Given a whole column, it does not know whether you mean "all rows", "any row", or "row by row", so it stops with `ValueError: The truth value of a Series is ambiguous`. "Ambiguous" means "it could mean several things".

– *Common mistake 2, missing brackets:* Python works out `&` **before** `==` and `>`. So in `df["Division_Name"] == "Consumer" & df["Actual_Expense_USD"] > df["Budget_Expense_USD"]`, it first tries `"Consumer" & df["Actual_Expense_USD"]`, which makes no sense, and raises a `TypeError`. Brackets force the right order: `(df["Division_Name"] == "Consumer") & (df["Actual_Expense_USD"] > df["Budget_Expense_USD"])`.

– In these examples, `is_over = df["Actual_Expense_USD"] > df["Budget_Expense_USD"]`.

– *More examples:*

Code	Result	What it tells you
`(df["Division_Name"] == "Logistics") & is_over`	`Q1`, `Q3`	the Logistics quarters that overspent
`(df["Fiscal_Quarter"] == "Q4") & is_over`	`Enterprise`	only one division overspent in Q4
`(df["Audit_Status"] == "Variance_Flag") & (df["Fiscal_Quarter"] == "Q3")`	`TXN-5008`, `TXN-5009`, `TXN-5011`	the flagged quarters in Q3
`(~is_over).sum()`	`8`	~ flips the rule: 8 quarters stayed within budget
`df["Division_Name"].isin(["Cloud", "Enterprise"]).sum()`	`8`	`.isin()` is a short way to write "Cloud or Enterprise"

- *Remember:* one rule gives one `True` / `False` per row; combine rules with `&`, `|`, or `~`, and put each rule in its own brackets.

- **7.2 Split-apply-combine: how `groupby()` builds a summary**
  - *The idea:* Most management questions are about groups: "How much did **each division** spend?" Answering them always takes the same three steps, whether in an Excel pivot table or in pandas:
    1. **Split** the rows into groups, one per division.
    2. **Apply** a calculation inside each group, such as a sum.
    3. **Combine** the answers into a new, smaller table with one row per group.
  - *See it on real data:* the Consumer division's 4 quarterly rows go into one group, and `"sum"` adds them up:

Step	Consumer division
Split	Q1 `810000.0`, Q2 `795000.0`, Q3 `825000.0`, Q4 `775000.0`
Apply (`"sum"`)	`810000.0 + 795000.0 + 825000.0 + 775000.0 = 3205000.0`
Combine	one row: `Consumer 3205000.0`

    The same steps run for Cloud, Enterprise, and Logistics at the same time, so the 16 rows become 4: Cloud `1785000.0`, Consumer `3205000.0`, Enterprise `1280000.0`, Logistics `2468000.0`.
  - *What you choose:* the column to split by (`groupby("Division_Name")`), the column to calculate on, and the calculation: `"sum"` for totals, `"mean"` for averages, `"count"` for how many rows, `"max"` for the largest value.
  - *More examples:*

Code	Result	What it tells you
`df.groupby("Fiscal_Quarter")["Actual_Expense_USD"].sum()`	Q1 `2185000.0`, Q2 `2171000.0`, Q3 `2235000.0`, Q4 `2147000.0`	total spending per quarter: Q3 was the most expensive
`df.groupby("Division_Name")["Transaction_ID"].count()`	`4` for each division	a quick check that no quarter is missing
`df.groupby("Cost_Center_Code")["Actual_Expense_USD"].max()`	`CC-202` `825000.0` (largest)	the most expensive quarter of each cost center
`df.groupby("Audit_Status")["Transaction_ID"].count()`	`Variance_Flag` `6`, `Verified` `10`	how many quarters were flagged

  - *Remember:* `groupby()` does the split, `.agg()` does the apply, and pandas does the combine for you.

- **7.3 Joining tables: match by name, not by position**

- *The idea:* Information often sits in two tables: spending in one, the responsible VP in another. Joining means bringing the matching row from the second table next to each row of the first, using a column that both share: the key column, here `Cost_Center_Code`.
- *Why Excel's `VLOOKUP` is fragile:* `VLOOKUP` is told to return "the 2nd column" of the lookup table. If someone later inserts a new column, such as `Region`, the 2nd column is now `Region`, and every formula quietly returns a region instead of a VP name. No error appears, so the mistake is easy to miss.
- *Why `pd.merge()` is safer:* `pd.merge(spending, vp_info, on="Cost_Center_Code")` matches rows by the **name** of the key column and brings over every other column by name. Moving or inserting columns changes nothing.
- In these examples, `ledger = pd.merge(df, vp_info, on="Cost_Center_Code", how="left")`, as in Micro-Example 3, and `vp_renamed` is `vp_info` with its key column renamed to `CostCenter`.
- *More examples:*

Code	Result	What it tells you
`ledger.shape`	`(16, 8)`	still 16 rows; one new column, `VP_Lead`, added (7 + 1)
`ledger.loc[ledger["Division_Name"] == "Logistics", "VP_Lead"]`	`David Kim` in all 4 rows	one VP row is matched to every quarter of that cost center
`pd.merge(df, vp_renamed, on="Cost_Center_Code")`	`KeyError: 'Cost_Center_Code'`	`vp_renamed` calls the key `CostCenter`: the names must match exactly

- *Remember:* name the key column with `on=`; the key must have exactly the same name and the same values in both tables.

- **7.4 What happens to rows with no match: `how="left"` vs `how="inner"`**
  - *The idea:* Sometimes a row in the main table has no partner in the other table, for example a new cost center that has not been given a VP yet. The `how=` setting decides what happens to that row.
  - *See it on an example:* suppose our spending table also had a new cost center, `CC-505`, which is not in `vp_info`:

`Cost_Center_Code`	`Variance_USD`	`VP_Lead` with `how="left"`	with `how="inner"`
`CC-202`	`85000.0`	Marcus Vance	Marcus Vance
`CC-404`	`28000.0`	David Kim	David Kim
`CC-505`	`5000.0`	`NaN` (empty)	row removed

    `NaN` stands for "Not a Number", pandas' mark for a missing value.
  - *Why it matters:* with `how="inner"`, the `CC-505` overspend silently disappears from the report, and nobody is asked to explain it. With `how="left"`, the row stays and the empty `VP_Lead` tells you that someone must be assigned.

- In these examples, `vp_3_rows` is `vp_info` without the `CC-404` row (as if Logistics had no VP yet), and `merged` is the result of the left merge.
- *More examples:*

Code	Result	What it tells you
`pd.merge(df, vp_3_rows, on="Cost_Center_Code", how="left")`	16 rows, 4 with `NaN` in `VP_Lead`	`vp_3_rows` lacks `CC-404`: its 4 quarters stay, with no VP
`pd.merge(df, vp_3_rows, on="Cost_Center_Code", how="inner")`	12 rows	the 4 Logistics quarters silently disappear
`merged["VP_Lead"].isnull().sum()`	4	count the gaps after a left merge

- *Remember:* use `how="left"` when every row of your main table must stay in the report, then look for `NaN` to find the gaps.

8. Development Workflow in Google Antigravity IDE

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S14_demo_1.py`), and check the dataset is in `data/`.
- `[2]` Code Editor: write a few lines, run, repeat. Small steps keep errors small.
- `[3]` AI Agent Chat: on an error, capture the traceback (Python's error report) with `Win + Shift + S` (macOS: `Cmd + Shift + 4`), paste it with `Ctrl + V`, and ask why it failed.
- `[4]` Terminal: run `python` followed by your script name. Return code `0` means the script finished with no errors.

9. Hands-On Coding Micro-Examples

- Every example has three parts: **1. Input**, **2. Process**, and **3. Output**. Name each file with your own name and the session code, for example `Reacher_S14_demo_1.py` (pattern: `YourName_S14_demo_1.py`).
- *Micro-Example 1 (Find Overspending with a Boolean Mask):* `YourName_S14_demo_1.py`

```python
# 1. Input: load the quarterly budget data
import pandas as pd
df = pd.read_csv("data/session_01_04_dataset.csv")

# 2. Process: Consumer quarters that spent more than budget
mask = (
    (df["Division_Name"] == "Consumer")
    & (df["Actual_Expense_USD"] > df["Budget_Expense_USD"])
)
overruns = df[mask]
```

```
# 3. Output: count and list them
print("Consumer overrun quarters:", len(overruns))
print(overruns[["Fiscal_Quarter", "Budget_Expense_USD",
                "Actual_Expense_USD"]])
```

  - *How it works, line by line:*
    - `(df["Division_Name"] == "Consumer")` gives one `True` / `False` per row: `True` for the 4 Consumer rows.
    - `(df["Actual_Expense_USD"] > df["Budget_Expense_USD"])` compares the two money columns row by row: `True` where a quarter spent more than planned.
    - `&` keeps `True` only where both are `True` (section 7.1). The result, `mask`, is a column of 16 `True` / `False` values. Naming it makes the rule easy to read and reuse.
    - `df[mask]` keeps the rows where `mask` is `True` and stores them in `overruns`.
    - `len(overruns)` counts those rows.
  - *Read the output:*

```
Consumer overrun quarters: 3
  Fiscal_Quarter  Budget_Expense_USD  Actual_Expense_USD
1             Q1            780000.0            810000.0
5             Q2            780000.0            795000.0
9             Q3            780000.0            825000.0
```

    - The Consumer division overspent in 3 of its 4 quarters; only Q4 (`775000.0`) stayed within the `780000.0` budget.
    - Q3 has the largest overrun: `825000 - 780000 = 45000`, or `5.77%` over budget.
    - The labels `1`, `5`, `9` are the original row positions in the 16-row file.

- *Quick Check 1 (~):*
  - New tool: ~ (not) turns every `True` into `False`, and every `False` into `True`. Why: find the opposite group, such as quarters that stayed within budget.
  - Task: in part **2. Process**, change `& (df["Actual_Expense_USD"] > df["Budget_Expense_USD"])` to `& ~(df["Actual_Expense_USD"] > df["Budget_Expense_USD"])`.
  - Check: the script prints `Consumer overrun quarters: 1`, and the table shows `Q4` (budget `780000.0`, actual `775000.0`). The label still says "overrun", since only the rule changed.

- *Micro-Example 2 (Total Each Division with `groupby()`):* `YourName_S14_demo_2.py`

```
# 1. Input: load the quarterly budget data
import pandas as pd
df = pd.read_csv("data/session_01_04_dataset.csv")

# 2. Process: yearly totals for each division
div_summary = df.groupby("Division_Name").agg(
    Total_Budget_USD=("Budget_Expense_USD", "sum"),
    Total_Actual_USD=("Actual_Expense_USD", "sum"),
    Quarters=("Transaction_ID", "count"),
```

```
)
div_summary["Variance_USD"] = (
    div_summary["Total_Actual_USD"]
    - div_summary["Total_Budget_USD"]
)

# 3. Output: one row per division
print(div_summary[["Total_Actual_USD", "Variance_USD"]])
```

  - *How it works, line by line:*
    - `df.groupby("Division_Name")` splits the 16 rows into 4 groups, one per division (section 7.2).
    - Inside `.agg(...)`, each line has the form `New_Name=("Source_Column", "calculation")`. For example, `Total_Budget_USD=("Budget_Expense_USD", "sum")` adds up the budget of each division and calls the result `Total_Budget_USD`.
    - `Quarters=("Transaction_ID", "count")` counts the rows in each group; it is a quick check that every division has 4 quarters.
    - The next line subtracts two of the new columns to get each division's yearly variance.
  - *Read the output:*

```
                Total_Actual_USD  Variance_USD
Division_Name
Cloud                 1785000.0      -15000.0
Consumer              3205000.0       85000.0
Enterprise            1280000.0           0.0
Logistics             2468000.0       28000.0
```

    - The division names are now the row labels: `groupby()` turns the group names into the index.
    - Consumer overspent by `85000.0` for the year and Logistics by `28000.0`; Enterprise spent exactly its budget; Cloud stayed `15000.0` under.
    - Consumer had one quarter under budget, yet it is still the largest yearly overspender: quarterly flags and yearly totals answer different questions.

- *Quick Check 2 (`.sort_values()`):*
  - New tool: `.sort_values()` reorders rows by one column; `ascending=False` puts the largest first. Why: the director reads the biggest overspender first.
  - Task: add one line at the end of part **2. Process**: `div_summary = div_summary._____(` `"Variance_USD", ascending=False)`
  - Check: the first row is `Consumer` (`85000.0`), then `Logistics` (`28000.0`).

- *Micro-Example 3 (Add VP Names with `pd.merge()`):* `YourName_S14_demo_3.py` (VP names are fictional)

```
# 1. Input: budget data and a VP list (fictional names)
import pandas as pd
df = pd.read_csv("data/session_01_04_dataset.csv")
vp_info = pd.DataFrame({
```

```
    "Cost_Center_Code": ["CC-101", "CC-202", "CC-303", "CC-404"],
    "VP_Lead": ["Sarah Chen", "Marcus Vance",
                "Elena Rostova", "David Kim"],
})

# 2. Process: add VP names, then flag spending over budget
ledger = pd.merge(df, vp_info, on="Cost_Center_Code", how="left")
ledger["Over_Budget"] = (
    ledger["Actual_Expense_USD"] > ledger["Budget_Expense_USD"]
)

# 3. Output: show the first three rows
print(ledger[["Transaction_ID", "VP_Lead", "Over_Budget"]].head(3))
```

  - *How it works, line by line:*
    - `pd.DataFrame({...})` builds a small table directly in code. Inside the curly brackets, each column name is followed by its list of values, in row order. The two lists must have the same length (here 4).
    - `pd.merge(df, vp_info, on="Cost_Center_Code", how="left")` takes each of the 16 rows of `df`, looks up its cost center in `vp_info`, and adds the matching `VP_Lead` (section 7.3). `how="left"` keeps every row of `df`, even one with no match (section 7.4).
    - `ledger["Actual_Expense_USD"] > ledger["Budget_Expense_USD"]` compares row by row and stores `True` / `False` in a new `Over_Budget` column.
  - *Read the output:*

```
  Transaction_ID        VP_Lead  Over_Budget
0       TXN-5000     Sarah Chen        False
1       TXN-5001   Marcus Vance         True
2       TXN-5002  Elena Rostova        False
```

    - Every row now carries the name of the person responsible for that spending; no VP name was copied by hand.
    - `TXN-5001` (Consumer, Q1) is over budget, and Marcus Vance is accountable for it.
    - The table still has 16 rows; `.head(3)` only shows the first 3.

- *Quick Check 3 (`.value_counts()`):*
  - New tool: `.value_counts()` counts how often each value appears in a column. Why: answers "how many quarters were over budget?" in one line.
  - Task: add one line to part **3. Output**: `print(ledger["Over_Budget"].____________())`
  - Check: the Terminal shows `False 8` and `True 8`: half of the 16 quarters went over budget.

10. Official Python Documentation & References

- Pandas User Guide: GroupBy: Split-Apply-Combine.
- Pandas User Guide: Merge, Join, Concatenate and Compare.
- Python PEP 8: Guidelines for Line Breaks around Binary Operators.

11. Exercise 1: Guided Practice - Quarterly Division Audit

- *Step 1: Open Script:* In Google Antigravity IDE, create a new script named `YourName_S14_audit.py`, replacing `YourName` with your own name (for example, `Reacher_S14_audit.py`).
- *Step 2: Group by Division and Quarter:* Type the following code:

```
# 1. Input: load the quarterly budget data
import pandas as pd
df = pd.read_csv("data/session_01_04_dataset.csv")

# 2. Process: variance per row, then group by two columns
df["Variance_USD"] = df["Actual_Expense_USD"] - df["Budget_Expense_USD"]
df["Variance_Pct"] = df["Variance_USD"] / df["Budget_Expense_USD"] * 100.0
quarterly_audit = df.groupby(["Division_Name", "Fiscal_Quarter"]).agg(
    Total_Actual=("Actual_Expense_USD", "sum"),
    Total_Variance=("Variance_USD", "sum"),
    Mean_Variance_Pct=("Variance_Pct", "mean"),
)

# 3. Output: one row per division and quarter
print("Quarterly Division Audit Summary:\n", quarterly_audit)
```

  - Grouping by a list of two columns gives one row per division and quarter: 16 rows, labeled by both values.

- *Step 3: Verify Output Integrity:* Confirm that quarters with overruns show positive `Total_Variance` values. 8 of the 16 rows should be positive.
- *Step 4: Execute in Terminal:* Run the script via Panel `[4]` (Terminal) using `python YourName_S14_audit.py` and confirm return code `0`.

12. Exercise 2: Applied Practice - Annual Cost Center Overruns

- *Coding Challenge:* In `YourName_S14_practice.py`, find the cost centers that overspent for the whole year.
- *Task 1 (Annual Cost Center Aggregation):* Group the data by `Cost_Center_Code`, sum budget and actual spending, compute the yearly variance %, and keep cost centers more than `1.0`% over budget.
- *Task 2 (Deliberate KeyError & AI Diagnosis):* Merge the result with `vp_info` (from Micro-Example 3), but type `on="Cost_Center"` on purpose. Read the `KeyError` (the column name does not exist), capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and review the AI's explanation of join keys.
- *Task 3 (Audit Brief Output):* Fix the key to `on="Cost_Center_Code"`, then print the cost center, the VP, and the overrun in USD.
- *Success Check:* 2 cost centers: `CC-202` (Marcus Vance, `85000.0`, 2.72%) and `CC-404` (David Kim, `28000.0`, 1.15%); the script ends with return code `0`.

13. Exercise 3: Advanced Practice - VP Exception Report

- *Goal:* Build an exception report that tells the finance director which VPs must explain their spending. A VP is escalated when the cost center has repeated flagged quarters **and** overspent for the whole year.
- *What you need:*
  - File: `data/session_01_04_dataset.csv` (16 rows, described in **5. Lab Practice Dataset**).
  - The `vp_info` table from Micro-Example 3 (VP names are fictional).
  - Script name: `YourName_S14_challenge.py`.
  - Tools you already know: Boolean masks, `groupby()` + `.agg()`, `pd.merge()`, `.sort_values()`, and `.to_csv()`.
- *Requirements (follow them in order):*
  1. Add `Variance_USD` = `Actual_Expense_USD - Budget_Expense_USD` and `Variance_Pct` = `Variance_USD` / `Budget_Expense_USD` x 100, rounded to 2 decimal places.
  2. Add `My_Flag` = `True` when `Variance_Pct > 2.0`. Check your rule: count the rows where `My_Flag` disagrees with `Audit_Status == "Variance_Flag"`. The count must be `0`.
  3. **Flagged summary:** keep only the flagged rows, then group by `Cost_Center_Code` and compute `Flagged_Quarters` (count of rows), `Flagged_Overrun_USD` (sum of `Variance_USD`), and `Worst_Variance_Pct` (largest `Variance_Pct`).
  4. **Yearly summary:** using all 16 rows, group by `Cost_Center_Code`, sum budget and actual spending, and compute `Year_Variance_Pct` = (actual - budget) / budget x 100, rounded to 2 decimal places.
  5. Start from `vp_info` and merge in the yearly summary, then the flagged summary, both with `on="Cost_Center_Code"` and `how="left"`. If a cost center had no flagged quarter, its empty values must become `0`.
  6. Add `Escalate` = `True` when `Flagged_Quarters >= 2` **and** `Year_Variance_Pct > 1.0`.
  7. Sort by `Flagged_Overrun_USD`, then by `Worst_Variance_Pct`, both largest first. Print the table and save it to `data/vp_exception_report.csv` without row labels.
- *New tools for this exercise:*
  - `.reset_index()`: after `groupby()`, the group names (here, the cost center codes) become the row labels. `.reset_index()` turns them back into a normal column, so you can merge on `Cost_Center_Code`.
  - `.fillna(0)`: replaces empty values (`NaN`) with `0`, for example `report["Flagged_Quarters"] = report["Flagged_Quarters"].fillna(0)`.
  - Sorting by two columns: `.sort_values(["Flagged_Overrun_USD", "Worst_Variance_Pct"], ascending=False)`. The second column only decides ties in the first.
  - `"max"` inside `.agg()` returns the largest value in each group, just as `"sum"` returns the total.
- *Hints:*
  - Build the two summaries as two separate tables first, print each one, and only then merge.
  - `(df["My_Flag"] != (df["Audit_Status"] == "Variance_Flag")).sum()` counts disagreements, because each `True` counts as 1.
- *Self-Check (expected results):*
  - Rule check: `0` disagreements; 6 of the 16 quarters are flagged.

- Table order: `CC-202` Marcus Vance (2 flagged, `75000.0`, worst `5.77`), `CC-404` David Kim (2, `45000.0`, `4.92`), `CC-303` Elena Rostova (1, `10000.0`, `3.12`), `CC-101` Sarah Chen (1, `10000.0`, `2.22`).
- `Escalate` is `True` only for Marcus Vance (`Year_Variance_Pct` `2.72`) and David Kim (`1.15`). Elena Rostova and Sarah Chen each had one flagged quarter but finished the year at or under budget.