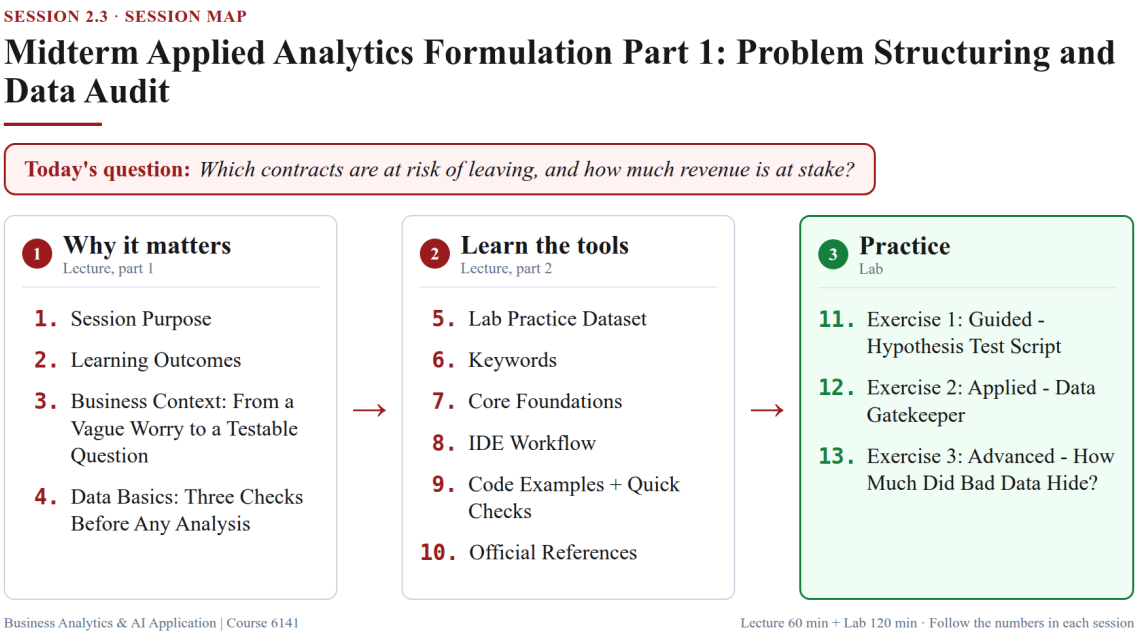
Session 2.3: Midterm Applied Analytics Formulation Part 1: Problem Structuring and Data Audit

Course 6141: Business Analytics & AI Application | Reading for Week 7



Schedule: Week 7 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- An IT (Information Technology) services company holds 15 enterprise client contracts worth $10.7 million a year
- The account director has a vague worry: “some clients are unhappy and may leave”, but a worry is not something data can test
- The account director asks: which contracts are at risk of leaving, and how much revenue is at stake? Today, you turn that worry into a testable question, check that the data can be trusted, and measure the starting point. This is step 1 of your midterm project

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

1. “What exactly should we test?”: turn a vague business problem into a testable hypothesis (a clear statement that data can support or prove wrong)

2. "Can we trust this data?": audit a dataset for missing values (`.isnull()`), duplicates (`.duplicated()`), and impossible values, such as a percentage above 100
3. "How do we stop bad data automatically?": write `assert` checks (rules the data must pass) and handle an `AssertionError` with `try` / `except`

3. Business Context: From a Vague Worry to a Testable Question

- Managers rarely hand you a clean question. A useful analysis moves through four steps:

Step	Example
Vague worry	"Some clients are unhappy and may leave."
Business question	"Is poor service linked to clients at risk of leaving?"
Testable hypothesis	"Accounts below 95% SLA compliance have more billing disputes and a higher retention risk than other accounts."
Measure today (baseline)	"Revenue in High-risk accounts today: $930,000."

- SLA (Service Level Agreement): the service promise in a contract, such as "the system is available 99% of the time". SLA compliance is the share of that promise actually delivered.
- Baseline: today's value of the measure you care about. Without it, nobody can prove later that an action improved anything.
- **Midterm Project, Step 1 of 2: your Question Card.** Choose a business problem with a dataset you can audit (for example, any dataset from Sessions 1.3 to 2.2), and fill in five lines:
    1. Vague worry: what the manager is concerned about
    2. Business question: what decision the answer supports
    3. Hypothesis: one sentence the data can support or prove wrong
    4. Data and audit: the file you use, and the result of the three checks in section 4
    5. Baseline: one number that describes today
- Bring your completed Question Card and your audit script to Session 2.4, where you present your findings.

4. Data Basics: Three Checks Before Any Analysis

- Bad data produces confident but wrong answers ("garbage in, garbage out"). Run three checks before you analyze:

Check	Question	pandas
Completeness	Are any values missing?	`df.isnull().sum()`
Uniqueness	Does each ID appear only once?	`df["Account_ID"].duplicated().sum()`

Check	Question	pandas
Validity	Are all values possible?	`(df["SLA_Compliance_Pct"] > 100).sum()`

- `NaN` ("Not a Number") is how pandas shows an empty cell. `.isnull()` marks each `NaN` as `True`.
- A rule for validity comes from business logic: a percentage must be between 0 and 100, and a contract value must be above 0.

5. Lab Practice Dataset

- Two versions of the same 15 client contracts (company names are fictional):
  - Clean file: `data/session_02_03_dataset.csv`, 15 rows that pass every check. Use it for analysis.
  - Raw file: `data/session_02_03_raw.csv`, 16 rows as they arrived from the billing system, with a few problems for you to find. Use it for auditing.
- Columns (the same in both files):
  - `Account_ID`: Unique client account code (e.g., `ACC-301`).
  - `Enterprise_Client`: Client company name (Titan Tech, Beacon Health, Vanguard Retail, and others). One company can hold several accounts.
  - `Contract_Value_USD`: Yearly contract value in U.S. Dollars.
  - `SLA_Compliance_Pct`: Share of the service promise delivered, in percent.
  - `Billing_Discrepancy_Count`: Number of billing disputes (invoices the client challenged) during the contract.
  - `Client_Retention_Risk`: The account team's rating of how likely the client is to leave: `Low`, `Medium`, or `High`.

6. Keywords (Core Programming Concepts)

- `Hypothesis`:
  - What: one clear statement that data can support or prove wrong
  - Why: turns a vague worry into a test with a yes or no answer
- `Exploratory Data Analysis (EDA)`:
  - What: a first look at the data with counts, averages, and charts before any formal test
  - Why: shows what the data can answer, and where it looks strange
- `Data Audit`:
  - What: the three checks: completeness, uniqueness, and validity
  - Why: one bad row can change a total by millions
- `Baseline`:
  - What: today's value of the measure you care about

  - Why: the starting point for proving any later improvement

- `assert`:
  - What: a line that stops the script with an `AssertionError` when a rule is broken, such as `assert (df["Contract_Value_USD"] > 0).all()`
  - Why: bad data is caught at the door, before it reaches a report

- `try` / `except`:
  - What: runs code in `try`; if a chosen error happens, the code in `except` runs instead of stopping the script
  - Why: the script can report the problem clearly and keep going

7. Core Programming & Computational Foundations

- **7.1 From a worry to a hypothesis you can test**
  - *The idea:* A hypothesis is one sentence that data can support or prove wrong. A useful hypothesis names what you measure, the groups you compare, and the difference you expect. "Some clients are unhappy" cannot be tested; "accounts below 95% SLA compliance have more billing disputes than the others" can, because the data has a column for each part.
  - *See it on real data:*

Part of the hypothesis	In our data
What is measured	`Billing_Discrepancy_Count`
The two groups	SLA below 95% (`4` accounts) vs 95% or above (`11` accounts)
The expected difference	the first group has more disputes
The result	`4.75` vs `0.64` disputes on average: supported

  - *Common mistake:* a hypothesis that cannot fail, such as "service quality matters". If no result could ever prove it wrong, it does not help anyone decide.
  - *More examples:*

Vague worry	Testable hypothesis	Column needed
"Our big clients are risky."	Accounts above $800,000 have a higher share of High risk ratings.	`Contract_Value_USD`, `Client_Retention_Risk`
"Billing is a mess."	Accounts with 3 or more disputes are rated Medium or High risk.	`Billing_Discrepancy_Count`, `Client_Retention_Risk`
"Service is slipping."	High-risk accounts have a lower average SLA than Low-risk accounts.	`SLA_Compliance_Pct`, `Client_Retention_Risk`

In our data, the third one is supported (High-risk accounts average `90.97`% SLA, Low-risk `98.84`%), while the first is not: none of the 6 accounts above $800,000 is rated High risk. A hypothesis that turns out wrong is still useful: it rules out a wrong explanation.

- *Remember:* a good hypothesis names the measure, the groups, and the expected difference.

- **7.2 The three data checks**
  - *The idea:* Before any analysis, check that the data is complete (no missing values), unique (no record counted twice), and valid (every value is possible). Each check is a rule on a column that gives `True` / `False` per row; `.sum()` counts the `True` values, because Python counts `True` as 1.
  - *See it on real data:* the raw file (16 rows).

Check	Code	Result	The problem
Completeness	`raw.isnull().sum().sum()`	`1`	`ACC-306` has no SLA value
Uniqueness	`raw["Account_ID"].duplicated().sum()`	`1`	`ACC-304` appears twice
Validity	`(raw["SLA_Compliance_Pct"] > 100).sum()`	`1`	`ACC-311` shows `105.0`%
Validity	`(raw["Contract_Value_USD"] <= 0).sum()`	`1`	`ACC-313` has a negative value

  - *Why it matters:* these 4 small problems change the totals. The raw file adds up to `$11,290,000` of contracts; the corrected file to `$10,710,000`. The average SLA looks like `97.45`% in the raw file but is really `96.48`%.
  - *Common mistake:* `.duplicated()` marks only the **second** copy as `True`. To see both copies of `ACC-304`, use `raw.duplicated("Account_ID", keep=False)`.
  - *More examples:*

Code	Result	What it tells you
`raw.isnull().sum()`	`1` in `SLA_Compliance_Pct`, `0` elsewhere	which column has the gap
`raw.loc[raw["SLA_Compliance_Pct"].isnull(), "Account_ID"]`	`ACC-306`	which account has the gap
`raw["Account_ID"].is_unique`	`False`	a one-word answer to “are the IDs unique?”
`raw.loc[raw["SLA_Compliance_Pct"] > 100, "Account_ID"]`	`ACC-311`	which account has an impossible percentage

Code	Result	What it tells you
`(~raw["SLA_Compliance_Pct"].between(0, 100)).sum()`	`2`	`.between()` treats a missing value as outside the range too

  - *Remember:* complete, unique, valid; check all three before you calculate.

- **7.3 Rules the data must pass: `assert`**
  - *The idea:* `assert rule, "message"` checks a rule. If the rule is `True`, nothing happens and the script continues. If it is `False`, the script stops with an `AssertionError` that shows your message. Rules turn your data checks into a gate: bad data cannot reach the report unnoticed.
  - *See it on real data:*

Code	On the raw file	On the clean file
`assert (df["Contract_Value_USD"] > 0).all(), "Contract value must be above 0"`	stops: `AssertionError: Contract value must be above 0`	passes
`assert df["Account_ID"].is_unique, "Account IDs must be unique"`	stops	passes

  - *`.all()` and `.any()`:* a rule on a column gives many `True` / `False` values, but `assert` needs one answer. `.all()` is `True` only if every value is `True`; `.any()` is `True` if at least one is.
  - *Common mistake:* leaving out the message. `assert rule` alone stops with an empty `AssertionError`, and the reader has to guess which rule failed.
  - *More examples:*

Rule	On the raw file	On the clean file
`assert (df["SLA_Compliance_Pct"] <= 100).all()`	fails (`105.0`, and a missing value counts as `False`)	passes
`assert len(df) == 15`	fails (16 rows)	passes
`assert df["Client_Retention_Risk"].isin(["Low", "Medium", "High"]).all()`	passes	passes
`assert not df.isnull().any().any()`	fails (1 gap)	passes

  - *Remember:* one `assert` per rule, always with a message.

- **7.4 Handling a failure: `try` / `except`**

- *The idea:* Sometimes you want the script to report the problem and keep going, for example to finish a cleaning step. Put the risky lines under `try:`. If an `AssertionError` happens, Python jumps to the lines under `except AssertionError as error:` instead of stopping, and `error` holds the message.

  ```
  try:
      assert (raw["Contract_Value_USD"] > 0).all(), "Contract value must be
  ↪   above 0"
  except AssertionError as error:
      print("Data problem:", error)
  ```

  On the raw file this prints `Data problem: Contract value must be above 0`, and the script continues.

- *Common mistake:* catching every error with a bare `except:`. It also hides typos and real bugs. Name the error you expect.

- *More examples:*

Code inside `try:`	Error raised	The matching `except` line
`int("abc")`	`ValueError: invalid literal for int() with base 10: 'abc'`	`except ValueError:`
`df["Missing"]`	`KeyError: 'Missing'`	`except KeyError:`
`1 / 0`	`ZeroDivisionError: division by zero`	`except ZeroDivisionError:`
`assert False, "Bad data"`	`AssertionError: Bad data`	`except AssertionError:`

- *Remember:* `try` the check, `except` the expected error, and report it clearly.

- **7.5 The baseline: today's number**

- *The idea:* A baseline is the value of your measure today. It is the starting point that any later improvement is compared with. Without it, nobody can prove that an action worked.

- *See it on real data:*

Baseline	Value today
Revenue in High-risk accounts	`$930,000`
Share of all revenue at High risk	`8.7%`
Accounts below 95% SLA	`4` of `15` (`26.7%`)
Billing disputes in total	`26`

  If the company improves service next quarter and the High-risk revenue falls to, say, $500,000, the baseline shows the size of the improvement.

- *Common mistake:* measuring the baseline on raw data. With the raw file, High-risk revenue looks like `$310,000` (a negative typo hides `$620,000`): any later "improvement" would be measured from a false start.

- *More examples:*

Code	Result	What it tells you
`df["SLA_Compliance_Pct"].mean()`	`96.48`	average service level today
`df["Billing_Discrepancy_Count"].mean()`	`1.73`	average disputes per account
`len(df[df["Client_Retention_Risk"] == "High"])`	`3`	accounts rated High risk
share of revenue in Low-risk accounts	`72.5%`	the safe part of the portfolio

  - *Remember:* audit first, then record the baseline.

8. Development Workflow in Google Antigravity IDE

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S23_demo_1.py`), and check both files are 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_S23_demo_1.py` (pattern: `YourName_S23_demo_1.py`).
- *Micro-Example 1 (Audit the Raw File):* `YourName_S23_demo_1.py`

```python
# 1. Input: the raw file, as it arrived
import pandas as pd
raw = pd.read_csv("data/session_02_03_raw.csv")

# 2. Process: the three checks, as counts
missing_values = raw.isnull().sum().sum()
duplicate_ids = raw["Account_ID"].duplicated().sum()
sla_above_100 = (raw["SLA_Compliance_Pct"] > 100).sum()
value_not_positive = (raw["Contract_Value_USD"] <= 0).sum()

# 3. Output: one line per check
print("Rows:", len(raw))
print("Missing values:", missing_values)
print("Duplicate account IDs:", duplicate_ids)
```

```
print("SLA above 100%:", sla_above_100)
print("Contract value of 0 or less:", value_not_positive)
```

  - *How it works, line by line:*
    - The script reads the **raw** file, as it came from the billing system, before anything is cleaned.
    - `raw.isnull()` marks every empty cell `True`. The first `.sum()` counts per column, and the second adds the columns together (section 7.2).
    - `raw["Account_ID"].duplicated()` marks a row `True` when its ID already appeared above; `.sum()` counts them.
    - `(raw["SLA_Compliance_Pct"] > 100)` and `(raw["Contract_Value_USD"] <= 0)` mark impossible values; `.sum()` counts them.
    - Part 3 prints one line per check, so the report reads like a checklist.
  - *Read the output:*

```
Rows: 16
Missing values: 1
Duplicate account IDs: 1
SLA above 100%: 1
Contract value of 0 or less: 1
```

    - The file has 16 rows, but only 15 real contracts: one row is a copy.
    - Each check finds exactly 1 problem, 4 in total. Section 7.2 shows which accounts they are.
    - None of these problems raises an error by itself: the file loads normally. Only the checks reveal them.

- *Quick Check 1 (`.drop_duplicates()`):*
  - New tool: `raw.drop_duplicates()` returns the table without rows that exactly repeat an earlier row. Why: a copied row counts the same contract twice.
  - Task: add one line to part **3. Output**: `print("Rows after removing duplicates:", len(raw._____())))`
  - Check: the Terminal shows `Rows after removing duplicates: 15`.

- *Micro-Example 2 (Test the Hypothesis on the Clean File):* `YourName_S23_demo_2.py`

```
# 1. Input: the clean file
import pandas as pd
df = pd.read_csv("data/session_02_03_dataset.csv")

# 2. Process: mark SLA breaches, then compare the two groups
df["SLA_Breach"] = df["SLA_Compliance_Pct"] < 95.0
comparison = df.groupby("SLA_Breach").agg(
    Accounts=("Account_ID", "count"),
    Avg_Disputes=("Billing_Discrepancy_Count", "mean"),
).round(2)

# 3. Output: breach (True) vs no breach (False)
print(comparison)
```

  - *How it works, line by line:*
    - The script reads the **clean** file: analysis is done on data that passed the checks.
    - `df["SLA_Compliance_Pct"] < 95.0` marks each account `True` (breach) or `False`, and stores the result as a new column, `SLA_Breach`.
    - `df.groupby("SLA_Breach")` splits the accounts into the 2 groups (Session 1.4). `.agg()` counts the accounts and averages the disputes in each group.
    - `.round(2)` keeps 2 decimals.
  - *Read the output:*

```
             Accounts  Avg_Disputes
SLA_Breach
False              11          0.64
True                4          4.75
```

    - The 4 accounts below the 95% promise average `4.75` billing disputes; the other 11 average `0.64`, about 7 times fewer.
    - The first part of the hypothesis is supported: poor service and billing disputes go together. Quick Check 2 tests the second part, the link with retention risk.

- *Quick Check 2 (`pd.crosstab()`):*
  - New tool: `pd.crosstab(column_a, column_b)` counts the rows for every pair of values, such as breach yes or no against risk Low, Medium, or High. Why: tests the second half of the hypothesis in one table.
  - Task: add one line to part **3. Output**: `print(pd._____(df["SLA_Breach"], df["Client_Retention_Risk"]))`
  - Check: in the `True` row, `High` is `3` and `Medium` is `1`: 3 of the 4 breached accounts are rated High risk, and no non-breached account is.

- *Micro-Example 3 (Measure the Baseline):* `YourName_S23_demo_3.py`

```python
# 1. Input: the clean file
import pandas as pd
df = pd.read_csv("data/session_02_03_dataset.csv")

# 2. Process: contract value by risk rating, and the High-risk share
value_by_risk = df.groupby("Client_Retention_Risk")["Contract_Value_USD"].sum()
high_risk_value = value_by_risk["High"]
high_risk_share = high_risk_value / df["Contract_Value_USD"].sum() * 100

# 3. Output: today's baseline
print(value_by_risk)
print("High-risk share of revenue (%):", round(high_risk_share, 1))
```

  - *How it works, line by line:*
    - `df.groupby("Client_Retention_Risk")["Contract_Value_USD"].sum()` adds up contract value for each risk rating. Choosing one column after `groupby()` gives a simple list of totals instead of a table.

    - `value_by_risk["High"]` reads one value from that list, by its label.
    - The share divides the High-risk total by all contract value and multiplies by 100.

  – *Read the output:*

```
Client_Retention_Risk
High       930000.0
Low       7770000.0
Medium    2010000.0
Name: Contract_Value_USD, dtype: float64
High-risk share of revenue (%): 8.7
```

    - `$930,000` a year, `8.7`% of revenue, sits in accounts rated High risk. This is the baseline (section 7.5).
    - Most revenue (`$7,770,000`) is in Low-risk accounts, so the business is not in immediate danger, but the 3 High-risk accounts are where action pays off first.
    - The last line of the list, `Name: ... dtype: float64`, is pandas describing the list: its name and its type. It is not data.

- *Quick Check 3 (f-string number format):*

  – New tool: `f"${value:,.0f}"` writes a number with thousands separators and no decimals, such as `$930,000`. Why: managers read money faster in this form.
  – Task: add one line to part **3. Output**: `print(f"Revenue at High risk: ${high_risk_value:___}")`
  – Check: the Terminal shows `Revenue at High risk: $930,000`.

10. Official Python Documentation & References

- Python Tutorial: Errors and Exceptions (`try` / `except`).
- Python Reference: The `assert` Statement.
- Pandas User Guide: Working with Missing Data.

11. Exercise 1: Guided Practice - Hypothesis Test Script

- *Step 1: Open Script:* In Google Antigravity IDE, create a new script named `YourName_S23_hypothesis.py`, replacing `YourName` with your own name (for example, `Reacher_S23_hypothesis.py`).
- *Step 2: Test Two Hypotheses and Measure the Baseline:* Type the following code:

```
# 1. Input: the clean file
import pandas as pd
df = pd.read_csv("data/session_02_03_dataset.csv")

# 2. Process: two tests and one baseline
# Hypothesis 1: higher-risk accounts have more billing disputes
disputes_by_risk = (
    df.groupby("Client_Retention_Risk")["Billing_Discrepancy_Count"]
```

```
    .mean().round(2)
)
# Hypothesis 2: lower SLA compliance goes with more disputes
sla_vs_disputes = df["SLA_Compliance_Pct"].corr(
    df["Billing_Discrepancy_Count"]
)
# Baseline: revenue in High-risk accounts today
high_risk_value = df.loc[
    df["Client_Retention_Risk"] == "High", "Contract_Value_USD"
].sum()

# 3. Output: a short report
print("Hypothesis 1: average disputes by risk rating\n", disputes_by_risk)
print(f"Hypothesis 2: correlation of SLA and disputes: {sla_vs_disputes:.3f}")
print(f"Baseline: revenue at High risk: ${high_risk_value:,.0f}")
```

  - `df.loc[rows, "Column_Name"]` selects the rows and one column at the same time.
  - `{sla_vs_disputes:.3f}` shows the number with 3 decimals.

- *Step 3: Interpret:* Write two comment lines at the end of the script: does each result support its hypothesis?
- *Step 4: Execute in Terminal:* Run `python YourName_S23_hypothesis.py` in Panel `[4]` and confirm return code `0`.
- *Success Check:* average disputes are `5.33` for High, `2.0` for Medium, and `0.25` for Low; the correlation is `-0.982` (lower SLA goes with more disputes); the baseline is `$930,000`. Both hypotheses are supported.

12. Exercise 2: Applied Practice - Data Gatekeeper

- *Coding Challenge:* In `YourName_S23_practice.py`, write checks that stop bad data, then clean the raw file so it passes.
- *Task 1 (Write the Rules):* Load `data/session_02_03_raw.csv`. Write three `assert` lines with messages:
  - `assert raw["SLA_Compliance_Pct"].between(0, 100).all(), "SLA must be between 0 and 100"`
  - `assert (raw["Contract_Value_USD"] > 0).all(), "Contract value must be above 0"`
  - `assert not raw["Account_ID"].duplicated().any(), "Account IDs must be unique"`
  - `.between(0, 100)` is `True` when a value is from 0 to 100; a missing value counts as `False`.
- *Task 2 (Read the AssertionError & AI Diagnosis):* Run the script. It stops at the first rule with `AssertionError: SLA must be between 0 and 100`. Capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and ask which rows break the rule.
- *Task 3 (Clean, Re-check, Save):* Put the three `assert` lines inside `try:` and print the message under `except AssertionError as error:`. Then build `clean` from `raw`: remove duplicate rows with `.drop_duplicates()`, keep only rows where the SLA is between 0 and 100, and keep only rows where

the contract value is above 0. Run the three checks again on `clean`, and save it to `data/contracts_clean.csv` with `index=False`.

- *Success Check:* the first run prints `SLA must be between 0 and 100`; `clean` has `12` rows and passes all three checks; the script ends with return code `0`.

13. Exercise 3: Advanced Practice - How Much Did Bad Data Hide?

- *Goal:* Show the account director, in numbers, how the raw file's problems distort the risk report, and list exactly which accounts the data team must fix.
- *What you need:*
  - Files: `data/session_02_03_raw.csv` (with problems) and `data/session_02_03_dataset.csv` (the corrected version of the same contracts).
  - Script name: `YourName_S23_challenge.py`.
  - Tools you already know: the three checks, `groupby()`, `.sum()`, Boolean masks with `|`, `.loc[]`, `pd.DataFrame()`, and f-strings.
- *Requirements (follow them in order):*
  1. Write a function `audit(table)` that returns a dictionary with four counts: `"missing"`, `"duplicate_ids"`, `"sla_out_of_range"` (SLA above 100), and `"value_not_positive"` (contract value of 0 or less).
  2. Call `audit()` on both files and print both dictionaries.
  3. Make a list of every `Account_ID` in the raw file that breaks at least one rule. Each account should appear only once, sorted from smallest to largest.
  4. For each file, sum `Contract_Value_USD` by `Client_Retention_Risk`. Put the two results side by side in one DataFrame with the columns `Raw_USD` and `Correct_USD`, and add `Difference_USD` = `Raw_USD` - `Correct_USD`.
  5. Print the table and one sentence that states the biggest distortion in plain words, using an f-string with thousands separators.
- *New tools for this exercise:*
  - Defining a function: `def audit(table):` followed by indented lines, ending with `return {...}`. Calling `audit(raw)` runs those lines on `raw` and gives back the dictionary.
  - `|` means "or" for masks: `(rule_1) | (rule_2)` is `True` when at least one rule is `True`. `.isnull()` on one column marks its missing values.
  - Unique, sorted values: `sorted(set(raw.loc[mask, "Account_ID"]))`. `set()` keeps each value once; `sorted()` puts them in order.
  - Side-by-side columns: `pd.DataFrame({"Raw_USD": raw_sums, "Correct_USD": correct_sums})`, where both sums come from `groupby()`, lines up the values by risk rating.
- *Hints:*
  - For requirement 3, build one mask that is `True` for a duplicate, a missing value, an SLA above 100, or a value of 0 or less. Use `raw.duplicated("Account_ID", keep=False)` so that both copies of a duplicated account are marked.
  - Print each intermediate result before you combine them.
- *Self-Check (expected results):*

- Raw file: `{"missing": 1, "duplicate_ids": 1, "sla_out_of_range": 1, "value_not_positive": 1}`; clean file: all four counts are `0`.
- Accounts to fix: `['ACC-304', 'ACC-306', 'ACC-311', 'ACC-313']`.
- Differences: `High -620000.0`, `Low 1200000.0`, `Medium 0.0`.
- Business meaning: one negative typo (`ACC-313`) made High-risk revenue look like `$310,000` instead of `$930,000`, hiding two-thirds of the real risk, while one duplicated row added `$1,200,000` to Low risk. The data must be fixed before anyone acts on the risk report.