

cdcu v1 first uploaded 29 Aug 2026
@System Fundamentals — v1 is work in progress
v2 updates soon

CDCU | Climate Data Capability Upgrade

► **Tool 1** — Temperature

Converter

► **Tool 2** — CDCU Jitter Tool

► **Tool 3** — Quartile & Quintile

Distribution Tester

► **Download Free Zip Bundle**

— PDF + 3 Python Scripts



CDCU: Structural Dataset Correction → New Analytical Capability

Climate Data Capability Upgrade = Problem → Demo → Fix → 3 Free Python Tools →

Concept: Physics conversion & micro-jittering to resolve a fundamental weakness in NOAA temperature °F datasets.

Problem: Public climate data can lack structural integrity for downstream ranking analysis and spatial encoding—coarse resolution and arbitrary zero points break balanced quantile distributions. This article is concerned exclusively with establishing structural ranking integrity, distinct from general statistical quantile analysis.

Response: Two straightforward methods can reestablish structure in a **1-year dataset**, creating compatibility with downstream ranking and spatial mapping systems. First, **lock the metric to fundamental physics by converting °F to Kelvin**—an absolute thermodynamic scale incapable of negative values. Second, **deploy variable bidirectional multiplicative micro-jitter to break ties**, turning a duplicate-heavy NOAA series into a stable numerical structure.

Reproducible Research: Delivered in the spirit of computational reproducibility (Open Data, Open Code, Executable Results), this write-up is paired with public Python tools to allow independent verification. CDCU is open-source and a work in progress. Published as a PDF with Python files, **the**

copy and paste Python code runs immediately on Google Colab. This allows users to investigate the Houston NOAA dataset, generate simulated ranges, or import their own CSVs to test failure modes, calibrate jitter, and observe bin distributions.

Disclaimer: There is nothing new in CDCU — it is simply an approach to a known data binning issue. Published as a free, open-source resource suitable for modification or adaptation, I am not qualified in academic theory and make no claim beyond practical utility. The current release is V1, with a V2 update due in the near future bringing formalised maths and improved code.

Common Methods for Handling Duplicate Values and Ties: A short AI-generated overview of standard tie-resolution practices is appended at the end (iv).

Data Source: <https://www.ncei.noaa.gov/access/services/data/v1?dataset=daily-summaries&stations=USW00012960&startDate=2024-01-01&endDate=2024-12-31&dataTypes=TMAX,TMIN,TAVG,ADPT&units=standard&format=csv>

3 x Open Source Python Tools

Tool 1: Temperature Converter Tool °C & °F → Kelvin (K)

Scope: Anchors interval-scale measurements °C & °F to an absolute thermodynamic zero K.

Function: Asks if the dataset is Celsius (°C) or Fahrenheit (°F), then converts to Kelvin (K).

Tool 2: Bidirectional Multiplicative Micro-Jitter Tool

Scope: Establishes unique / improved addressability by breaking numerical ties caused by excessive duplications within a coarse dataset.

Jitter Mechanism: Multiplicative factor = $1.0 \pm \text{offset}$ | Active Offset Range: $\pm(0.00000001 \text{ to } 0.00000999)$ | Precision: 5 static leading zeros + 3 active digits (001–999)

Function: Imports a dataset and applies micro-jitter to the range. This creates micro-distortions to the original value elevating the structural integrity of the data for downstream processing. The process is slightly destructive.

Tool 3: Quartile & Quintile Distribution Tester

Scope: Illustrates results using industry-standard code to show if the dataset has enough structure to test on more advanced ordinal or spatial systems.

Function: Imports the dataset with or without jitter, running a simultaneous quartile (4-bin) and quintile (5-bin) split. Using industry-standard Python Pandas, the terminal outputs a comparative diagnostic. **It demonstrates whether the dataset passed or failed the Quartile & Quintile Distribution Test—passing because it successfully met the equal-frequency test with evenly distributed bins, or failing due to tied clusters straddling cut-off boundaries.** Displays count, high and low, cut-off points and can be exported as CSV.

CDCU | Climate Data Capability Upgrade

Public data can be free, widely used, and still require structural preparation before serious statistical analysis.

NOAA, the National Oceanic and Atmospheric Administration, provides some of the most widely relied-upon public weather and climate data in the world. Its datasets support climate research, insurance, agriculture, logistics, and countless other forms of analysis. But widely used does not necessarily mean analysis-ready. A dataset can contain accurate observations while its published representation limits what can realistically be done with that data. The problem may be an arbitrary zero point, weak resolution, or excessive duplication caused by rounding.

This article examines a 1-year NOAA dataset from Houston Intercontinental Airport — a time-series of daily, non-negative temperature readings in °F — and attempts to resolve a problem with basic mathematics. Now, one question: Can basic mathematics make a public dataset with known weaknesses structurally more robust, leading to potential industry application?

The First Problem — Not All Metrics Have an Absolute Zero

Before any analysis, it helps to distinguish between an absolute physical metric and measurements expressed on human-created scales. Absolute thermodynamic temperature and mass represent fundamental classes of physical measurement that possess a true absolute zero. Most other scales do not have a natural absolute zero; while perfectly valid and useful, their zero points are established by convention rather than by the complete absence of the measured quantity.

Fahrenheit: Is a historical temperature scale whose human-defined zero was based on the freezing point of a saltwater brine mixture, resulting in a degree size that differs from the modern SI temperature scale. Developed alongside the mercury thermometer in 1714 and finalised as a scale in 1724, it subsequently became the primary temperature standard for the British Empire.

Celsius: Is a practical temperature scale built around the freezing and boiling points of water. A Celsius degree has the same size as a Kelvin degree, but zero Celsius is not an absolute physical zero.

Kelvin: Is the SI base unit for thermodynamic temperature. Zero Kelvin corresponds to absolute zero, the physical lower limit of thermodynamic temperature.

This distinction matters because mathematical operations inherit the properties of the scale used as their input. Celsius and Fahrenheit are interval scales. They are entirely legitimate for reporting and

comparing temperature differences, but their raw numerical values cannot safely be treated as ratio-scale quantities.

20°C Is Not Twice as Hot as 10°C — this is the simplest demonstration — it is sometimes casually stated that 20°C is twice as hot as 10°C — it isn't. Convert both values to an absolute temperature scale: 20°C = 293.15 K

10°C = 283.15 K — The ratio is: $293.15 / 283.15 \approx 1.035$

So on an absolute thermodynamic scale, 20°C is only about 3.5% higher in absolute temperature than 10°C. Another issue can cause confusion when identical numerical labels occur on different scales.

A London summer day at 20°C is: 293.15 K

A New York winter day at 20°F is: 266.48 K

Both are labelled "20°", but their absolute temperatures differ by approximately 26.67 K (equivalent in °C). The numbers are comparable only after the underlying scale has been understood. This is not a defect in Celsius or Fahrenheit. They were designed for practical human use. The problem occurs when their numerical values are fed into mathematical operations that assume an absolute zero.

The Second Problem — Resolution & Distribution

"Temperature is not a perfectly stable quantity". Outdoor air is continuously changing, and attempting to represent it to extreme decimal precision does not necessarily produce more useful information. Some variation is inherent in the physical environment and in the measurement process itself. But the opposite problem can also occur: published resolution can become so coarse that meaningful variation is compressed into the same numerical value. Consider this sample NOAA data from Houston Intercontinental Airport Station USW00012960:

date	dew-point-f	avg-temp-f	TMAX	TMIN
2024-01-06	42	52	64	40
2024-01-07	42	50	62	38
2024-01-08	54	63	76	50
2024-01-09	29	48	56	40
2024-01-10	40	51	69	33

The observations are published as whole degrees Fahrenheit. Across a full year, this produces a relatively small set of possible integer values and therefore a large number of duplicate values (ties). That matters when the dataset is subsequently ranked, as these ties must be resolved.

Why Resolution Matters for Quantile Ranking

Ordinal Ranking: Orders observations from low to high.

Quantile Ranking: Takes that ordered distribution and divides it into equal-frequency buckets or sections; these are called bins. In practice, quantile ranking serves two distinct applications. The first allows for ties and tolerates tie imbalance, which is perfectly fine for localized analysis. The second requires strictly balanced, equal distributions per bin count to maintain structural integrity for quantile ranking — plus potentially spatial mapping / telemetry. This article deals exclusively with the structural requirements for quantile ranking analysis.

Common examples include: Quartiles — four bins | Quintiles — five bins | Percentiles — one hundred bins

A dataset with sufficient resolution can distribute observations evenly through the ranking structure. If that data package has concentrations of identical values, it cannot always do so. For example, if a year's observations repeatedly contain the same rounded temperature, those observations occupy the same numerical position. A ranking method may therefore have to place a large group of identical observations on one side of a quantile boundary rather than distributing them evenly across it. While mathematically valid for general statistics, this result violates the strict structural constraints required for spatial mapping.

The problem goes deeper because the published dataset's resolution lacks the distinct numerical positions required to fulfil a strictly balanced distribution across the range. This means the dataset's structural integrity for equal binning is compromised before downstream processing even begins. Below shows examples of ranked 1-year datasets, with different active days counts being tested for distribution Pass / Fail integrity.

The Data Resolution to Distribution Rule

✓ Pass	X Fail
365 = 73,73,73,73,73	366 = 71,73,75,74,73
250 = 50,50,50,50,50	257 = 50,52,53,51,51
304 = 61,61,61,61,60	323 = 63,65,66,64,65
301 = 61,60,60,60,60	275 = 53,55,56,57,54

Fail cases shown are purely illustrative. A fail pattern of this type occurs when the dataset fails to distribute evenly. For a 250-day dataset viewed through a quintile set, we need exactly 50 observations per bin. The distribution breaks when a cluster of duplicate values straddles a bin's cut-off boundary, forcing the formula to push the entire tie into a single bucket.

Quantile Definition: A quantile is the general term for any value dividing a dataset into equal intervals; quartiles (4 parts) and quintiles (5 parts) are specific types of quantiles.

The Bin Count Constraint: The distribution rule for this type of analysis is — **all bin counts must be equal or differ by a maximum of 1.**

Resolution / Structural Balance Requirement: The selection of a specific quantile type requires sufficient data resolution to establish an even distribution. For example the NOAA dataset can establish even quintile distribution with 2 active jitter digits, while the same dataset requires 3 active jitter digits for an even quartile split.

Resolution as a Quality Metric: Resolution quality can be assessed by counting decimal places. Two-decimal financial prices on Yahoo (e.g., close_price \$97.46) offer 100× more distinct values than whole-integer weather readings on NOAA (e.g., 97°F). The resolution difference is massive and seems to be the reason financial data can be used in complex analytical systems so well.

Jitter Precision Scaling: Fewer bins often means higher-resolution jitter is required, so be aware of **Jitter Precision Scaling**. The jittering may need to increase by additional digits — each added decimal place increases jitter precision tenfold. For example, a previous jitter of **0.00000273** may need to become **0.000002738**. The active digits are a 3-digit “**2•7•3**” and a 4-digit “**2•7•3•8**” randomizer at the micro scale.

1-Year Dataset Utility: Daily observations within a 1-year dataset (~366 points) have provided ample resolution for quartiles or quintiles, yielding bins with a mean of ~91 or ~73 items each. This is not necessarily true for percentile sets.

Elevating the Dataset

Two different structural problems have now been identified with the NOAA data which require two different treatments:

- The zero point is not physically absolute.
- The published resolution is too coarse for structured analysis.

Fix 1: Convert the Temperature to Kelvin

The first step is to move the temperature onto an absolute physical scale.

- **Celsius to Kelvin** $K = ^\circ C + 273.15$
- **Fahrenheit to Kelvin** $K = (^\circ F - 32) \times 5/9 + 273.15$

This does not create new information. It changes the numerical representation so that the data is transformed and anchored to a true physical zero. This provides several benefits:

- Absolute temperature is non-negative
- Ratio operations become physically meaningful
- Proportional calculations can use a valid zero reference
- The data is aligned with the underlying physical quantity

One Kelvin has the same magnitude as one Celsius degree, so Celsius-to-Kelvin conversion changes only the zero point. Fahrenheit-to-Kelvin conversion changes both the zero point and the degree size, because a Fahrenheit degree is 5/9 of a Kelvin. The conversion to Kelvin solves the zero-point problem but does not address the resolution problem.

Fix 2: Bidirectional Multiplicative Micro-Jitter

Scope: Establishes unique / improved addressability by breaking numerical ties caused by excessive duplications within a coarse dataset.

Jitter Mechanism: Multiplicative factor = $1.0 \pm \text{offset}$ | Active Offset Range: $\pm(0.00000001 \text{ to } 0.00000999)$ | Precision: 5 static leading zeros + 3 active digits (001–999)

Function: Imports a dataset and applies micro-jitter to the range. This creates micro-distortions to the original value elevating the structural integrity of the data for downstream processing. The process is slightly destructive.

Bidirectional Multiplicative Micro-Jitter. The multiplier range is an engineer's calibration choice, set according to the dataset and the precision the downstream process requires — there is no single correct range. **The jitter range here is 0.99999001 to 1.00000999 (about $\pm 0.001\%$),** sufficient to break up identical whole-degree Fahrenheit or Kelvin tie-clusters across both quartile (4-bin) and quintile (5-bin) configurations.

The example **micro-jitter consists of 5 zeros then factors at 3-digit active randomization (± 001 to ± 999 suffixes across 8 decimal places): 0.99999237, 1.00000713, 0.99999842, 1.00000195.** The randomization can be deployed at 3-, 4-, 5-, or 6-digit depth after the zeros, depending on dataset length and boundary tie density. Higher precision jitter creates more unique discrete state channels, improving overall structural balance.

Micro jitter factors shift the observed values marginally upward or downward. The objective is to use a seeded, reproducible micro-jitter as a deterministic tie-breaker. By separating these identical states, it reestablishes downstream processing integrity with minimal distortion to original published values.

Example jitter at value 51°F on a given day: $51 \times 0.99999237 = 50.99961087$ and $51 \times 1.00000713 = 51.00036363$ (note that both resulting values round cleanly back to 51). The published dataset values are preserved on the surface, while internally gaining the unique addressability required to break ties and maintain zero-deviation equal-frequency balance.

Why multiplicative? It applies a balanced bidirectional perturbation—distributing positive and negative micro-distortions proportionally across each non-negative integer except zero. This preserves the structural balance of the distribution while establishing the unique addressability required for downstream processing. For whole-degree climate data, the perturbation is placed several decimal places inward (at the millionths to hundred-millionths level).

Zero Handling (Additive Jitter): Multiplicative jitter doesn't work on zeros, to correct excessive zeros in a dataset, a similar function called Additive Micro-Jitter can be deployed. This is simply adding a tiny random number to the zero, $0 + 0.00000619 = 0.00000619$.

Three Analytical Capabilities Become Unlocked:

Once the temperature is bound to absolute zero using Kelvin, the doors immediately open to different forms of analysis. Note that datasets also require sufficient resolution to function correctly.

1. **Cardinal Magnitude:** The first benefit is valid proportional mathematics. Anchoring to an absolute physical zero gives ratios true physical meaning—meaning 300 K is genuinely twice 150 K—allowing the numerical value to be treated as a true magnitude rather than an arbitrary position on an interval scale.
2. **Ordinal Ranking:** The second benefit is that the dataset now has a valid ranking distribution; it passed the test as a consequence of the micro-jitter reducing duplicates. This distribution is a requirement for quantile analysis. The dataset has gained usable internal resolution, although some duplicates may still occur. Note that it is harder to achieve a valid distribution with a smaller number of quantile bins, such as quartiles (4 bins). This is more difficult to evenly distribute in comparison to quintiles (5 bins) and may require a finer jitter.
3. **Deeper Jitter Insights:**
 - **Basic ordinal/quantile distribution analysis** — sufficient ordinal resolution is needed for time-indexed ordinal data analysis. Duplicate clusters can be tolerated partially across quantile boundaries. An active set requires even-frequency bin distribution.
 - **Advanced ordinal/quantile analysis and telemetry “spatial encoding”** — substantially higher uniqueness and resolution may be required, potentially including observation-level identity/UIDs, where individual records require unique addressability within a multidimensional coordinate space.

A dataset that began as a collection of coarse, duplicated public measurements can therefore become a stable numerical structure capable of supporting additional forms of analysis and representation.

Standard Workarounds and Their Structural Trade-offs

When coarse resolution breaks equal-frequency binning, common approaches include dropping duplicate bin edges or applying flat additive noise to separate ties. Both resolve the immediate error but introduce structural trade-offs: positional tie-breaking decouples the distribution from the underlying measurements, while flat additive noise can push near-zero values negative — breaking physical reality for non-negative metrics. The approach described in this article addresses the same problem by working with the physical properties of the data rather than around them.

The Broader Lesson

Fahrenheit and Celsius are standard measurements with interesting structural imperfections — but the NOAA temperature data remains immensely useful. A whole-number temperature observation may be perfectly adequate for a weather report while initially failing a high-resolution ranking system.

Ultimately, public datasets are not finished analytical products; they are raw material. By identifying structural limitations early, we can engineer mathematical transformations that fundamentally elevate the data without destroying the original observations. It is not about changing the facts; it is about upgrading the underlying architecture so the dataset can actually support ordinal ranking and advanced statistical capabilities.

Four Added Points — Related Context

(i) NOAA Dataset limitations: The National Oceanic and Atmospheric Administration publishes a vast range of public weather and climate products that often use different representations or resolutions—for example, the global GHCN-Daily uses tenths of degrees Celsius, while US reporting stations normally present data in whole degrees Fahrenheit. The latter is low-resolution and can break the distribution for downstream processing. This experiment required structural modifications to a 1-year dataset so that downstream ordinal analysis could take place. NOAA could up its game and publish datasets with °F in tenths of degrees rather than integers, and also add the true physics temperature in K with any °C or °F datasets.

(ii) Online industry services exist to automatically apply many types of mathematical transformations for large datasets; however, high quality structural corrections can be executed in-house using a standard spreadsheet or Python code.

(iii) Ground Zero Constant Offset: Related technique for near-zero physical metrics such as wind or wave, distinct from the Kelvin conversion used above: When a dataset's values naturally rest near zero, proportional or percentage-based scaling becomes unstable — a day-to-day move from 0.1 mph to 50 mph wind speed, for instance, produces a percentage change of roughly 49,900%, which is mathematically correct but practically meaningless, since percentage change is undefined or explosive as the denominator approaches zero. This is a known limitation of metrics like MAPE (mean absolute percentage error), which is well documented as unreliable near zero and has motivated alternatives such as sMAPE and MASE; a comparable fix appears in log-transform workflows in finance and bioinformatics, where a small constant (a "pseudocount") is added before transformation to avoid the same singularity. The offset applied here follows that same logic: it shifts the effective baseline away from true zero, establishing a buffer between zero and the data's resting range, so proportional comparisons are calculated against a stable reference point and reflect real-world change rather than artifacts of a near-zero denominator.

(iv) Common Methods for Handling Duplicate Values and Ties

Note: This is an AI-generated section for general context following basic research—this is not drawn from specialist industry expertise.

The following four approaches are commonly used to resolve duplicate values (ties) in numerical datasets. They generally fall into two structural categories: **State-Dependent resolution** (keeping the exact original numbers but relying on how the system organizes them) and **Payload-Mutation** (slightly altering the data itself to force uniqueness).

- a. **Row-Order / Index Ranking (`rank(method='first')` or `ROW_NUMBER()`)** — Breaks ties based on the data's ingestion sequence. This preserves the exact original numbers, but it introduces a dependency on the system state: if the background sorting or ingestion order changes, the tie-breaking results will shift with it.
- b. **Flat Additive Jitter ($\pm\Delta$)** — Adds or subtracts a static micro-value to each tied number to force them apart. This method requires careful boundaries to avoid creating invalid data—for example, accidentally pushing a value of 0 into negative territory for metrics that can never be negative. Also as a side comment, the term "Flat Additive Jitter ($\pm\Delta$)" feels misleading as the effect is Flat Adjustive Jitter ($\pm\Delta$)?
- c. **Bidirectional Multiplicative Jitter ($x \times (1 \pm \epsilon)$)** — Applies a tiny percentage-based adjustment that scales with the size of the number. This safely prevents positive values from accidentally crossing below zero, but it mathematically fails to separate exact zeros (since any percentage of zero remains zero).
- d. **Targeted Zero-Additive Jitter ($0 + \delta$)** — A targeted fallback that adds a microscopic positive amount specifically to exact zeros, covering the blind spot of the multiplicative method. It successfully forces separation, but introduces a minor distortion by artificially lifting absolute zeros narrowly off the floor.

CDCU NOAA Experiment Results So Far

NOAA Dataset °F → Convert K (Tool 1) → Apply Jitter (Tool 2) → Test Distribution PASS / FAIL (Tool 3)

Below is copied from the Colab terminal using Tool 3. It shows the test results for the 2024 NOAA leap-year dataset (366 days), comparing equal-frequency binning performance across two runs: with and without micro-jitter.

— NOAA Houston Data With & Without Jitter After Kelvin Conversion

Tool 3: Quartile & Quintile Distribution Tester – WITH JITTER > ± 0.00000999
Equal-frequency binning diagnostic (4-bin and 5-bin)

=====

==

INPUT:

- File 'distro_data.csv' MUST be in the Colab working directory.
- Column A: label/date/notes (optional, max 12 chars).
- Column B: numeric metric to test.
- No CSV export in this draft.

TYPE 'reset' AT ANY PROMPT TO CLEAR AND RESTART.

=====

==

Loading 'distro_data.csv' ...

=====

==

STAGE 1: DATASET OVERVIEW

=====

==

Active rows: 366 / 366 (100.0%)

Minimum: 273.707557

Maximum: 312.041902

=====

==

STAGE 2: EQUAL-FREQUENCY DISTRIBUTION TEST

=====

==

--- QUANTILES (4 bins) ---

Cut-off points (0%, 25%, 50%, 75%, 100%):

\[273.70755699504, 297.59203710046404, 302.0415111262905, 306.8989402569052
3, 312.041901814218\]

Bin counts:

Q1: 92

Q2: 91

Q3: 91

Q4: 92

DISTRIBUTION RULE: PASS (ideal = 91.5, max deviation = 0.5)

--- QUINTILES (5 bins) ---

Cut-off points (0%, 20%, 40%, 60%, 80%, 100%):

\[273.70755699504, 295.924965011676, 300.9274990722, 304.263978310006, 307.594658379296, 312.041901814218\]

Bin counts:

Q1: 74

Q2: 73

Q3: 73

Q4: 73

Q5: 73

DISTRIBUTION RULE: PASS (ideal = 73.2, max deviation = 0.8)

=====
==

Type 'reset' to clear and start again: reset

Resetting Tool 3...

=====
==

Tool 3: Quartile & Quintile Distribution Tester – WITHOUT JITTER
Equal-frequency binning diagnostic (4-bin and 5-bin)

=====
==

INPUT:

- File 'distro_data.csv' MUST be in the Colab working directory.
- Column A: label/date/notes (optional, max 12 chars).
- Column B: numeric metric to test.
- No CSV export in this draft.

TYPE 'reset' AT ANY PROMPT TO CLEAR AND RESTART.

=====
==

Loading 'distro_data.csv' ...

=====
==

STAGE 1: DATASET OVERVIEW

=====

```

==
Active rows: 366 / 366 (100.0%)
Minimum: 273.705600
Maximum: 312.038900

=====
==
STAGE 2: EQUAL-FREQUENCY DISTRIBUTION TEST
=====
==

--- QUANTILES (4 bins) ---
Cut-off points (0%, 25%, 50%, 75%, 100%):
  \[273.7056, 297.5944, 302.0389, 306.9, 312.0389\]
Bin counts:
  Q1: 100
  Q2: 84
  Q3: 90
  Q4: 92
DISTRIBUTION RULE: FAIL (ideal = 91.5, max deviation = 8.5)

--- QUINTILES (5 bins) ---
Cut-off points (0%, 20%, 40%, 60%, 80%, 100%):
  \[273.7056, 295.9278, 300.9278, 304.2611, 307.5944, 312.0389\]
Bin counts:
  Q1: 81
  Q2: 79
  Q3: 60
  Q4: 82
  Q5: 64
DISTRIBUTION RULE: FAIL (ideal = 73.2, max deviation = 13.2)

=====
==

```

3 x Open-Source Python Tools:

These open-source Python tools are functional scripts under active development. The code includes inline comments to make it easy to adapt for larger datasets, different physics, industry metrics, or time intervals. Feel free to play around with the code and test it out on your own data. The tools accept datasets at 1-year, meaning simple CSV imports and exports with up to 366 rows.

Tool 1: Temperature Converter Tool °C & °F → Kelvin (K)

Tool 2: Bidirectional Multiplicative Micro-Jitter Tool

Tool 3: Quartile & Quintile Distribution Tester

cdcu.system-fundamentals.com

simon@system-fundamentals.com

Tool 1: Temperature Converter Tool °C & °F → Kelvin (K)

A. Theoretical Grounding (Interval vs. Ratio Scales)

Standard public climate records utilize human-defined interval scales (Celsius or Fahrenheit). Lacking a true physical zero, these scales prevent mathematically valid proportional or ratio-scale calculations (e.g., stating that 20 degrees C is "twice as hot" as 10 degrees C is mathematically invalid).

Converting these values to Kelvin (K) — the SI base unit for thermodynamic temperature — anchors the dataset to absolute physical zero. This establishes a structurally robust ratio-scale needed for some quantile and spatial mapping systems, enabling valid downstream proportional calculations and preventing mathematical scaling distortions.

B. Dataset Parameters (The Gate)

Colab file name: `temperature_data.csv`

Column A (Date or Label)

- Type: Text (optional)
- Maximum 12 characters
- Accepts dates, short identifiers, or can be left blank
- Blocks formulas, URLs, markdown image tags, and active code syntax

Column B (Active Metric)

- Type: Numeric (mandatory)
- Numbers only, no empty cells or gaps
- Supports negative values down to absolute zero
- Celsius: not below -273.15 degrees
- Fahrenheit: not below -459.67 degrees
- Maximum 10 characters (missing, needed v2)

Column C (Output)

- Type: Numeric (generated)
- Populates converted Kelvin values rounded to 4 decimal places
- Rejects any value below 0 Kelvin

Row Gate

- Exactly one header row

- Minimum 1 data row
- Maximum 366 data rows (single calendar year only)

C. NOAA Ingestion & Midpoint Fallback

To run real-world tests, download the 2024 daily summary dataset for Houston Intercontinental Airport (Station USW00012960) directly from **NOAA.gov** data. The URL can also be modified for different datasets:

<https://www.ncei.noaa.gov/access/services/data/v1?dataset=daily-summaries&stations=USW00012960&startDate=2024-01-01&endDate=2024-12-31&dataTypes=TMAX,TMIN,TAVG,ADPT&units=standard&format=csv>

Handling Missing (TAVG): This is the raw CSV file prior to Kelvin conversion—a Fahrenheit (°F) dataset where the daily average (TAVG) is missing, but daily highs (TMAX) and lows (TMIN) are provided. Before uploading, update the data table for (TAVG) using a standard formula, or proceed with (TMAX). $\text{Mean Temperature} = (\text{TMAX} + \text{TMIN}) / 2$

```

import sys, os, random, re
from datetime import datetime
import pandas as pd

# =====
# CDCU Tool 1: Temperature Converter (°C / °F to Kelvin) – 1-Year Dataset
# Climate Data Capability Upgrade | cdcu.system-fundamentals.com
# System-Fundamentals.com | Public Beta Open Code Test
# =====
# PURPOSE:
# Standard public climate temperature records use human-defined interval scales
# (°Celsius or °Fahrenheit). These scales lack a true physical zero, meaning their
# zero points are set by convention rather than by the complete absence of thermal
# energy. This makes proportional or ratio-scale calculations on raw values invalid.
#
# This tool converts temperature data to Kelvin (K) – the SI base unit for
# thermodynamic temperature – which anchors the dataset to absolute physical zero.
# This creates a structurally robust ratio-scale dataset, enabling valid downstream
# proportional calculations and preventing mathematical scaling distortions in
# advanced coordinate mapping and spatial partitioning pipelines.
#
# INPUT: A 2-column CSV file – Date | Raw Temperature (°C or °F)
# OUTPUT: A 3-column table with Column C populated as converted Kelvin values.
#
# DATASET RULES:
# - Exactly one header row followed by a minimum of 1 and maximum of 366 data rows.
# - Column A: Optional date/label. Max 12 characters. Blocked: formulas, URLs, images, code.
# - Column B: Mandatory numeric temperature values. No blanks. Negative values supported.
# - Column C: Auto-generated Kelvin output. Validated: no values below 0 K (absolute zero).
# =====

def clear_and_header():
    """
    Clears the terminal screen so the user always sees a clean, readable display,
    then prints a welcome banner with the tool name, project context, and NOAA
    data source URL so beginners can immediately fetch real-world climate data.
    """
    # os.name checks the operating system.
    # 'nt' means Windows (uses 'cls'), all others (Linux/macOS/Colab) use 'clear'.
    os.system('clear' if os.name != 'nt' else 'cls')
    print("=" * 80)
    print("CDCU Tool 1: Temperature Converter (°C / °F to Kelvin) – 1-Year Dataset")
    print("Climate Data Capability Upgrade | cdcu.system-fundamentals.com")

```

```

print("=" * 80)
print()
print("NOAA TEST DATA SOURCE:")
print("https://www.ncei.noaa.gov/access/services/data/v1?dataset=daily-summaries")
print("&stations=USW00012960&startDate=2024-01-01&endDate=2024-12-31")
print("&dataTypes=TMAX,TMIN,TAVG,ADPT&units=standard&format=csv")
print()
print("DATASET RULES:")
print("- Column A: Date or label (optional). Max 12 characters.")
print("- Column B: Raw temperature values (°C or °F). No blanks.")
print("- Column C: Auto-generated Kelvin output.")
print("- Row limit: 1 header + maximum 366 data rows (1 calendar year).")
print()
print("TYPE 'reset' AT ANY PROMPT TO CLEAR AND RESTART.")
print("TO STOP EXECUTION: click the stop button or delete the cell.")
print("=" * 80)
print()

```

```

def get_input(prompt):
    """
    A custom wrapper around Python's standard input() function.
    Does two things:
    1. Removes accidental leading/trailing spaces with .strip() so 'y ' works like 'y'.
    2. Detects 'reset' – raises a KeyboardInterrupt caught in main() to restart cleanly.
    Note: 'exit' is intentionally NOT handled; use Colab's stop button to end execution.
    """
    val = input(prompt).strip()
    val_lower = val.lower()

    if val_lower == 'reset':
        # Raising KeyboardInterrupt("RESET") is the signal we catch in main()
        # to clear all variables from memory and restart the program loop.
        raise KeyboardInterrupt("RESET")

    return val

def prompt_reset_only():
    """
    Repeatedly prompts the user to type 'reset' to restart the tool.
    Ignores any other input, including 'exit'.
    """
    print()
    while True:

```

```
get_input("Type 'reset' to clear and start again: ")
```

```
def generate_dummy():
    """
    Generates a synthetic 365-day dataset of Fahrenheit temperatures for testing.
    This allows anyone to immediately test the full tool without uploading a file.

    Uses 2025 – a standard non-leap year – so pd.date_range from Jan 1 to Dec 31
    produces exactly 365 rows with no extra handling required.

    Temperature range: 20°F to 80°F (whole integers, simulating coarse NOAA data).

    Returns:
        A pandas DataFrame with three columns:
        Date (YYYY-MM-DD), Raw_Temp_F (integer), Kelvin (float, 4 decimal places).
    """
    print()
    print("Generating 365-day synthetic Fahrenheit dataset (20°F to 80°F)...")
    print()

    # pd.date_range creates a sequence of calendar dates.
    # 2025 is a standard non-leap year: Jan 1 to Dec 31 = exactly 365 days.
    dates = pd.date_range(start="2025-01-01", end="2025-12-31", freq="D")

    # random.randint(20, 80) picks a whole number between 20°F and 80°F for each day.
    # This simulates the coarse whole-integer resolution common in public NOAA data.
    raw = [random.randint(20, 80) for _ in range(365)]

    # Convert each Fahrenheit value to Kelvin using the standard physical formula:
    # K = (°F - 32) × (5/9) + 273.15
    # Rounded to 4 decimal places for clean, consistent output.
    kelvin = [round((f - 32) * (5.0 / 9.0) + 273.15, 4) for f in raw]

    # Wrap into a pandas DataFrame – think of this as a virtual spreadsheet.
    # d.strftime("%Y-%m-%d") converts Python date objects into clean text strings.
    return pd.DataFrame({
        "Date": [d.strftime("%Y-%m-%d") for d in dates],
        "Raw_Temp_F": raw,
        "Kelvin": kelvin
    })

def validate_and_convert(path):
    """
```

Validates and converts a user-uploaded CSV file (2 columns) to Kelvin.

Validation checks performed in order:

1. File exists and is readable.
2. File has exactly 2 columns and between 1 and 366 data rows.
3. Column A values pass security checks (no formulas, URLs, images, or code).
4. Column B contains only valid numeric temperature values (no blanks).
5. User selects the input scale: Celsius (1) or Fahrenheit (2).
6. Kelvin conversion is applied.
7. Post-conversion check: no Kelvin values fall below absolute zero (0 K).

Args:

path: File path string to the user's CSV file.

Returns:

A pandas DataFrame with columns [Date, Raw_Temp, Kelvin] if valid.

Returns None if any validation step fails, prompting the user to retry.

"""

Load the file

try:

df = pd.read_csv(path, header=0)

except FileNotFoundError:

print(f"\n[Error: File not found at '{path}'. Please check the path and try again.]")

return None

except Exception as e:

print(f"\n[Error: Could not read file - {e}]")

return None

Column count check: require exactly 2 columns

if df.shape[1] != 2:

print(f"\n[Error: Expected exactly 2 columns, but found {df.shape[1]}].")

print("Required layout: Column A (Date/Label) | Column B (Temperature).")

return None

col_a, col_b = df.columns[0], df.columns[1]

Row count check: must have between 1 and 366 data rows (1 calendar year maximum)

row_count = len(df)

if row_count < 1:

print("\n[Error: The file contains no data rows. At least 1 row is required.]")

return None

if row_count > 366:

print(f"\n[Error: File contains {row_count} data rows. Maximum allowed is 366 (1

```

calendar year).]")
    return None

# Column A security validation
# Column A is optional but must pass all security checks if populated.
# This protects against formula injection, URLs, image files, and code injection.
for idx, val in enumerate(df[col_a]):
    val_str = str(val).strip()

    # Skip empty or NaN cells – these are allowed in Column A
    if val_str in ("", "nan", "None"):
        continue

    # Block cells exceeding 12 characters
    if len(val_str) > 12:
        print(f"\n[Error: Row {idx + 2}, Column A value '{val_str}' exceeds 12
characters. Max is 12.]")
        # idx + 2 accounts for the 0-based index + 1 for the header row
        return None

    # Block spreadsheet formula injection (cells starting with '=')
    if val_str.startswith("="):
        print(f"\n[Error: Row {idx + 2}, Column A contains a formula ('{val_str}').
Formulas are strictly blocked.]")
        return None

    # Block web URLs
    if "http://" in val_str or "https://" in val_str:
        print(f"\n[Error: Row {idx + 2}, Column A contains a URL ('{val_str}'). URLs are
strictly blocked.]")
        return None

    # Block image file references and markdown image tags
    if any(ext in val_str.lower() for ext in [".png", ".jpg", ".jpeg", ".gif"]) or "![\"
in val_str:
        print(f"\n[Error: Row {idx + 2}, Column A contains an image reference
('{val_str}'). Images are strictly blocked.]")
        return None

    # Block code syntax keywords
    code_indicators = ["import ", "def ", "lambda", "<script"]
    if any(indicator in val_str for indicator in code_indicators):
        print(f"\n[Error: Row {idx + 2}, Column A contains code syntax ('{val_str}').

```

```

Code elements are strictly blocked.])")
    return None

# Column B numeric validation: every cell must contain a valid numeric temperature
df["_metric"] = pd.to_numeric(df[col_b], errors="coerce")
if df["_metric"].isnull().any():
    bad_rows = df[df["_metric"].isnull()].index.tolist()
    print(f"\n[Error: Column B contains non-numeric or blank values at rows: {[r + 2 for
r in bad_rows]}].]")
    print("Every cell in Column B must contain a valid numeric temperature value.")
    return None

# Unit selection: user declares whether Column B is Celsius or Fahrenheit
print()
while True:
    unit = get_input(
        "Input Unit Selection:\n"
        "  1: Celsius (°C)\n"
        "  2: Fahrenheit (°F)\n"
        "Select [1 or 2]: "
    )
    if unit in ["1", "2"]:
        break
    print("Invalid choice. Please enter 1 for Celsius or 2 for Fahrenheit.")

# Kelvin conversion: Celsius to Kelvin or Fahrenheit to Kelvin
if unit == "1":
    df["Kelvin"] = df["_metric"] + 273.15
    unit_label = "°C"
else:
    df["Kelvin"] = (df["_metric"] - 32) * (5.0 / 9.0) + 273.15
    unit_label = "°F"

# Post-conversion thermodynamic boundary check: no Kelvin values below absolute zero
if (df["Kelvin"] < 0).any():
    bad_rows = df[df["Kelvin"] < 0].index.tolist()
    print(f"\n[Error: Calculated Kelvin values fall below absolute physical zero (0 K) at
rows: {[r + 2 for r in bad_rows]}].]")
    print(f"Please verify your Column B values are valid {unit_label} temperatures.")
    return None

# Package and return clean 3-column DataFrame ready for printing and optional CSV export
return pd.DataFrame({

```

```

        "Date": df[col_a].fillna(""),          # Fill empty date cells with blank string for
clean output
        "Raw_Temp": df[col_b],
        "Kelvin": df["Kelvin"].round(4)       # 4 decimal places for professional presentation
    })

```

```

def main():
    """
    Main program orchestrator.
    Manages the continuous loop, user flow, output printing,
    CSV export, and clean reset handling.
    """
    while True:
        try:
            # Clear screen and display header
            clear_and_header()

            # Input mode selection: dummy or uploaded CSV
            print("SELECT INPUT MODE:")
            print(" 1: Generate dummy 365-day dataset")
            print(" 2: Use 'temperature_data.csv' already uploaded to Colab")
            print()
            while True:
                mode = get_input("Enter 1 or 2: ")
                if mode in ["1", "2"]:
                    break
                print("Invalid choice. Please enter 1 or 2.")

            # Load data based on selected mode
            if mode == "1":
                df = generate_dummy()
            else:
                csv_filename = "temperature_data.csv"
                print(f"\nLooking for '{csv_filename}' in the Colab root directory...")
                df = validate_and_convert(csv_filename)
                if df is None:
                    # Validation failed: show reset prompt
                    print("\nPlease upload a corrected 'temperature_data.csv' file, or type
'reset' to restart.")
                    prompt_reset_only()
                    continue

            # Print the converted table

```

```

# Tab-delimited output for easy copy/paste into spreadsheets
print()
print("=" * 80)
print("CONVERTED TABLE – COPY AND PASTE READY")
print("=" * 80)
print("Date\t\tRaw_Temp\tKelvin")
print("-" * 80)
for _, row in df.iterrows():
    print(f"{row.iloc[0]}\t\t{row.iloc[1]}\t\t{row.iloc[2]}")
print("=" * 80)

# Print processing metadata: row count and timestamp
row_count = len(df)
timestamp_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
print(f"Rows Processed: {row_count} | Timestamp: {timestamp_str}")
print("=" * 80)
print()

# Optional CSV export
while True:
    save = get_input("Save output as CSV? (y/n): ")
    if save.lower() in ["y", "n"]:
        break
    print("Invalid input. Please enter 'y' or 'n'.")

if save.lower() == "y":
    while True:
        out_name = get_input("Enter filename (e.g., converted_kelvin.csv): ")
        out_name = out_name.strip()

        if out_name:
            break
        print("Filename cannot be empty.")
    try:
        df.to_csv(out_name, index=False)
        print()
        print(f"Success: File saved as '{out_name}' in your current directory.")
    except Exception as e:
        print(f"\n[Error saving file: {e}]")

print()

# Prompt user to reset (only option after successful run)
prompt_reset_only()

```

```

except KeyboardInterrupt as ki:
    if str(ki) == "RESET":
        # Clean reset: clear the dataframe from memory and restart the main loop
        print()
        print("Resetting CDCU Tool 1...")
        df = None # Removes the dataframe from active memory
        continue # Returns to the top of the while True loop

# If a generic interrupt occurs (e.g., user pressed stop), exit gracefully
print()
print("Interrupted. Stopping execution.")
sys.exit(0)

if __name__ == "__main__":
    # Standard Python entry point.
    # Ensures main() only runs when the script is executed directly,
    # not when it is imported as a module by another script.
    main()

```

Tool 2 – Bidirectional Multiplicative Micro-Jitter Tool

Scope: Establishes unique / improved addressability by breaking numerical ties caused by excessive duplications within a coarse dataset.

Jitter Mechanism: Multiplicative factor = $1.0 \pm \text{offset}$ | Active Offset Range: $\pm(0.00000001 \text{ to } 0.00000999)$ | Precision: 5 static leading zeros + 3 active digits (001–999)

Function: Imports a dataset and applies micro-jitter to the range. This creates micro-distortions to the original value elevating the structural integrity of the data for downstream processing. The process is slightly destructive.

This tool applies a fixed bidirectional multiplicative micro-jitter to a numeric column in a 2-column CSV. The input file must be named **jitter_data.csv** and pre-uploaded to the Colab working directory.

- **Column A (label / date / notes):** Passed through unchanged.
- **Column B (numeric metric):** Used as the input value.
- **Two new columns are generated:**
 - Multiplicative factor: a random value between 0.99999001 and 1.00000999.
 - Jittered value: rounded to 8 decimal places.
- **Fixed seed:** 42, for reproducible randomness.
- **Output:** Entire dataset printed as a tab-delimited table with 4 columns, copy tables in the code terminal.
- **Save option:** Export the file as CSV.
- **Jitter magnitude:** Microscopic (five zeros followed by a three-digit random suffix).
- **Purpose:** Creates just enough perturbation in coarse integer or Kelvin data to sustain equal-sized bin distributions for quartile and quintile testing, while preserving original ranking and physical meaning.

```

import sys, os, random
import pandas as pd
from datetime import datetime

# =====
# Tool 2: Bidirectional Micro-Jitter ( $\pm 0.00000999$ ) - Active Range
# Input: 'jitter_data.csv' (2 cols: label, metric) in Colab working dir
# Output: 4 cols (label, metric, factor, jittered) - all rows printed
# =====

def clear_and_header():
    os.system('clear' if os.name != 'nt' else 'cls')
    print("=" * 80)
    print("Tool 2: Bidirectional Micro-Jitter (fixed  $\pm 0.00000999$  magnitude)")
    print("MVP - quick tie-breaking for quartile and quintile testing")
    print("=" * 80)
    print("\nINPUT:")
    print(" - File 'jitter_data.csv' MUST be in the Colab working directory.")
    print(" - Column A: label/date/notes (optional, max 12 chars).")
    print(" - Column B: numeric metric to be jittered.")
    print(" - Jitter factor =  $1.0 \pm \text{random}(0.00000001 \text{ to } 0.00000999)$ .")
    print(" - Deterministic seed (42) for reproducibility.")
    print("\nTYPE 'reset' AT ANY PROMPT TO CLEAR AND RESTART.")
    print("=" * 80, "\n")

def get_input(prompt):
    val = input(prompt).strip()
    if val.lower() == 'reset':
        raise KeyboardInterrupt("RESET")
    return val

def prompt_reset_only():
    while True:
        get_input("Type 'reset' to clear and start again: ")

def validate_col_a(df, col_a):
    """Column A: if populated, must be  $\leq 12$  characters (anything allowed)."""
    for idx, val in enumerate(df[col_a]):
        if pd.isna(val):
            continue
        val_str = str(val).strip()
        if val_str == "":
            continue

```

```

        if len(val_str) > 12:
            return False, f"Row {idx+2} exceeds 12 chars: '{val_str}'"
    return True, ""

def apply_jitter(df, col_b):
    rng = random.Random(42)
    factors = []
    jittered = []
    for val in df[col_b]:
        try:
            num = float(val)
        except:
            num = float('nan')
        suffix = rng.randint(1, 999)          # 1..999
        offset = suffix / 100_000_000.0      # 0.00000001..0.00000999
        sign = 1 if rng.random() < 0.5 else -1
        factor = 1.0 + sign * offset
        factors.append(factor)
        jittered.append(num * factor)
    df['Factor'] = factors
    df['Jittered'] = jittered
    return df

def main():
    while True:
        try:
            clear_and_header()
            filename = "jitter_data.csv"
            print(f>Loading '{filename}' ...")
            try:
                df = pd.read_csv(filename, header=0)
            except FileNotFoundError:
                print(f"\n[Error: '{filename}' not found. Please upload it and try again.]")
                prompt_reset_only()
                continue
            except Exception as e:
                print(f"\n[Error reading file: {e}]")
                prompt_reset_only()
                continue

            if df.shape[1] < 2:
                print(f"\n[Error: Expected at least 2 columns, found {df.shape[1]}.]")
                prompt_reset_only()

```

```

        continue

col_a, col_b = df.columns[0], df.columns[1]

# Validate Column A (just length check)
ok, err = validate_col_a(df, col_a)
if not ok:
    print(f"\n[Column A validation failed: {err}]")
    print("Please correct the file and try again.")
    prompt_reset_only()
    continue

df = apply_jitter(df, col_b)

# Print entire table (all rows)
print("\n" + "=" * 80)
print("JITTERED TABLE (all rows)")
print("=" * 80)
print(f"{col_a}\t\t{col_b}\t\tFactor\t\tJittered")
print("-" * 80)
for _, row in df.iterrows():
    col_a_val = str(row[col_a]) if pd.notna(row[col_a]) else ""
    factor_str = f"{row['Factor']:.10f}"
    jit_str = f"{row['Jittered']:.10f}" if pd.notna(row['Jittered']) else "NaN"
    print(f"{col_a_val}\t\t{row[col_b]}\t\t{factor_str}\t\t{jit_str}")
print("=" * 80)
print(f"Total rows: {len(df)} | Timestamp: {datetime.now().strftime('%Y-%m-%d
%H:%M:%S')}}")
print("=" * 80, "\n")

# Export prompt
while True:
    save = get_input("Save the 4-column dataset as CSV? (y/n): ")
    if save.lower() in ['y', 'n']:
        break
    print("Please enter 'y' or 'n'.")
if save.lower() == 'y':
    while True:
        out_name = get_input("Enter filename (e.g., jittered_output.csv):
").strip()

        if out_name:
            break
        print("Filename cannot be empty.")

```

```

    try:
        df.to_csv(out_name, index=False)
        print(f"\nSuccess: File saved as '{out_name}'.")
    except Exception as e:
        print(f"\n[Error saving: {e}]")

prompt_reset_only()

except KeyboardInterrupt as ki:
    if str(ki) == "RESET":
        print("\nResetting Tool 2...")
        df = None
        continue
    else:
        print("\nStopped.")
        sys.exit(0)

if __name__ == "__main__":
    main()

```

Tool 3 – Quartile & Quintile Distribution Tester

Scope: Audits equal-frequency binning performance across Quartile (4-bin) and Quintile (5-bin) distributions for non-negative numerical data. This tool evaluates structural balance on 1-year datasets ranging from 1 to 366 active rows. It requires a 2-column CSV named **distro_data.csv** pre-uploaded to the Colab working directory. Col A is optional and can have dates, numbers, notes or leave blank, with the active Col B accepting non-negative values. The script executes a three-stage diagnostic audit:

1. Dataset Overview: Validates active row counts (1–366 rows), checks headers, and reports min/max values.
2. Boundary Calculation: Uses `np.percentile` to calculate quartile and quintile cut-off points.
3. Equal-Frequency Audit: Applies those cut-offs using `pd.cut`, reports bin occupancy, and issues a PASS/FAIL verdict against the Distribution Rule (max deviation ≤ 1.0 from ideal).

Note: Custom CSV export functionality and expanded code commentary are currently in development and will be included in the upcoming code update.

```

import sys, os
import numpy as np
import pandas as pd
from datetime import datetime

# =====
# Tool 3: Quartile & Quintile Distribution Tester
# Purpose: Validate if a dataset has sufficient resolution for equal-frequency
#          binning at both quartile (4-bin) and quintile (5-bin) levels.
# Input:   'distro_data.csv' (2 cols: label, metric) in Colab working dir
# Output:  Terminal diagnostic only - no CSV export.
# =====

def clear_and_header():
    os.system('clear' if os.name != 'nt' else 'cls')
    print("=" * 80)
    print("Tool 3: Quartile & Quintile Distribution Tester")
    print("Equal-frequency binning diagnostic (4-bin and 5-bin)")
    print("=" * 80)
    print("\nINPUT:")
    print("  - File 'distro_data.csv' MUST be in the Colab working directory.")
    print("  - Column A: label/date/notes (optional, max 12 chars).")
    print("  - Column B: numeric metric to test.")
    print("  - No CSV export in this draft.")
    print("\nTYPE 'reset' AT ANY PROMPT TO CLEAR AND RESTART.")
    print("=" * 80, "\n")

def get_input(prompt):
    val = input(prompt).strip()
    if val.lower() == 'reset':
        raise KeyboardInterrupt("RESET")
    return val

def prompt_reset_only():
    while True:
        get_input("Type 'reset' to clear and start again: ")

def validate_col_a(df, col_a):
    """Column A: if populated, must be ≤12 characters (anything allowed)."""
    for idx, val in enumerate(df[col_a]):
        if pd.isna(val):
            continue
        val_str = str(val).strip()

```

```

        if val_str == "":
            continue
        if len(val_str) > 12:
            return False, f"Row {idx+2} exceeds 12 chars: '{val_str}'"
    return True, ""

def stage1_overview(df, col_b):
    """Print dataset overview: row count / 366, min, max."""
    n = len(df)
    print("\n" + "=" * 80)
    print("STAGE 1: DATASET OVERVIEW")
    print("=" * 80)
    print(f"Active rows: {n} / 366 ({n/366*100:.1f}%)")
    vals = pd.to_numeric(df[col_b], errors='coerce')
    print(f"Minimum: {vals.min():.6f}")
    print(f"Maximum: {vals.max():.6f}")

def run_distribution_test(vals, q, label):
    """
    Run the equal-frequency distribution test for a given quantile (q).
    q = 4 for quartiles, q = 5 for quintiles.
    Returns: (edges, counts, max_deviation, pass_fail)
    """
    n = len(vals)
    if n == 0:
        return None, None, None, "No data"

    # Percentiles: q+1 points from 0% to 100%
    percentiles = [i/q * 100 for i in range(q + 1)]
    edges = np.percentile(vals, percentiles, method='linear')

    # Label bins
    bin_labels = [f"Q{i+1}" for i in range(q)]

    try:
        binned = pd.cut(vals, bins=edges, include_lowest=True, labels=bin_labels)
        counts = binned.value_counts().sort_index()
        ideal = n / q
        max_dev = max(abs(c - ideal) for c in counts)
        # PASS if all bins differ by ≤1 from ideal
        pass_fail = "PASS" if max_dev <= 1 else "FAIL"
        return edges, counts, max_dev, pass_fail
    except Exception as e:

```

```

        return edges, None, None, f"ERROR: {e}"

def stage2_distribution_test(df, col_b):
    """
    Stage 2: Run both quartile (4-bin) and quintile (5-bin) tests.
    Display cut-offs, bin counts, and PASS/FAIL.
    """

    vals = pd.to_numeric(df[col_b], errors='coerce').dropna()
    n = len(vals)
    if n == 0:
        print("\nNo valid numeric data.")
        return

    print("\n" + "=" * 80)
    print("STAGE 2: EQUAL-FREQUENCY DISTRIBUTION TEST")
    print("=" * 80)

    # Test 1: Quartiles (q=4)
    print("\n--- QUARTILES (4 bins) ---")
    edges, counts, max_dev, result = run_distribution_test(vals, q=4, label="quartile")

    if counts is not None:
        print("Cut-off points (0%, 25%, 50%, 75%, 100%):")
        print(f" {edges.tolist()}")
        print("Bin counts:")
        for label, count in counts.items():
            print(f" {label}: {count}")
        ideal = n / 4
        print(f"DISTRIBUTION RULE: {result} (ideal = {ideal:.1f}, max deviation = {max_dev:.1f})")
    else:
        print(f"Test failed: {result}")

    # Test 2: Quintiles (q=5)
    print("\n--- QUINTILES (5 bins) ---")
    edges, counts, max_dev, result = run_distribution_test(vals, q=5, label="quintile")

    if counts is not None:
        print("Cut-off points (0%, 20%, 40%, 60%, 80%, 100%):")
        print(f" {edges.tolist()}")
        print("Bin counts:")
        for label, count in counts.items():
            print(f" {label}: {count}")

```

```

        ideal = n / 5
        print(f"DISTRIBUTION RULE: {result} (ideal = {ideal:.1f}, max deviation =
{max_dev:.1f})")
    else:
        print(f"Test failed: {result}")

def main():
    while True:
        try:
            clear_and_header()
            filename = "distro_data.csv"
            print(f>Loading '{filename}' ...")
            try:
                df = pd.read_csv(filename, header=0)
            except FileNotFoundError:
                print(f"\n[Error: '{filename}' not found. Please upload it and try again.]")
                prompt_reset_only()
                continue
            except Exception as e:
                print(f"\n[Error reading file: {e}]")
                prompt_reset_only()
                continue

            if df.shape[1] < 2:
                print(f"\n[Error: Expected at least 2 columns, found {df.shape[1]}].")
                prompt_reset_only()
                continue

            col_a, col_b = df.columns[0], df.columns[1]

            ok, err = validate_col_a(df, col_a)
            if not ok:
                print(f"\n[Column A validation failed: {err}]")
                print("Please correct the file and try again.")
                prompt_reset_only()
                continue

            stage1_overview(df, col_b)
            stage2_distribution_test(df, col_b)

            print("\n" + "=" * 80)
            prompt_reset_only()

```

```
except KeyboardInterrupt as ki:
    if str(ki) == "RESET":
        print("\nResetting Tool 3...")
        df = None
        continue
    else:
        print("\nStopped.")
        sys.exit(0)

if __name__ == "__main__":
    main()
```