Data Validation & Quality

Week 2 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

Part 1: The Motivation

What did we actually collect?

Last Week: We Collected Data!

Remember our Netflix movie prediction project?

```python
# We wrote this beautiful code
movies = []
for title in movie_list:
    response = requests.get(OMDB_API, params={"t": title})
    movies.append(response.json())

df = pd.DataFrame(movies)
df.to_csv("netflix_movies.csv")
print(f"Collected {len(df)} movies!")
```

Output: `Collected 1000 movies!`

Feeling: Victory! Time to train models!

Reality Check: Let's Look at the Data

```python
import pandas as pd
df = pd.read_csv("netflix_movies.csv")
print(df.head())
```

```
   Title          Year    Runtime    imdbRating   BoxOffice
0  Inception      2010    148 min    8.8          $292,576,195
1  Avatar         2009    162 min    7.9          $760,507,625
2  The Room       2003    99 min     3.9          N/A
3  Inception      2010    148 min    8.8          $292,576,195
4  Tenet          N/A     150 min    7.3          N/A
```

Wait... something's wrong here.

The Problems Emerge

```
DATA QUALITY ISSUES

1. DUPLICATES:    Inception appears twice (rows 0 and 3)
2. MISSING:       Year is "N/A" for Tenet (row 4)
3. WRONG TYPES:   Runtime is "148 min" not integer 148
4. INCONSISTENT:  BoxOffice has "$" and commas
5. N/A VALUES:    Some BoxOffice entries are literally "N/A"
```

Let's Dig Deeper

```python
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Title       1000 non-null   object    ← All strings!
 1   Year        987 non-null    object    ← String, not int!
 2   Runtime     1000 non-null   object    ← "148 min" string
 3   imdbRating  892 non-null    object    ← String, not float!
 4   BoxOffice   634 non-null    object    ← "$292,576,195" string
```

Every column is a string (object)!
366 movies have no BoxOffice data!

What Happens If We Ignore This?

```python
# Naive approach: just train the model!
from sklearn.linear_model import LinearRegression

X = df[['Year', 'Runtime', 'imdbRating']]
y = df['BoxOffice']

model = LinearRegression()
model.fit(X, y)
```

```
ValueError: could not convert string to float: '148 min'
```

The model refuses to train.

Or Worse: Silent Failures

```python
# "Fix" by forcing numeric conversion
df['Year'] = pd.to_numeric(df['Year'], errors='coerce')
df['Rating'] = pd.to_numeric(df['imdbRating'], errors='coerce')

# Now 13 movies have NaN year, 108 have NaN rating
# We lost data silently!

# Train anyway
model.fit(df[['Year', 'Rating']].dropna(), y.dropna())
# Model trains on 521 movies instead of 1000!
```

You trained on half your data without realizing.

Real-World Data Quality Disasters

Company	What Happened	Cost
NASA Mars Orbiter	Metric vs Imperial units	$125 million
Knight Capital	Bad data in trading algorithm	$440 million in 45 min
UK COVID Stats	Excel row limit (65,536)	16,000 missing cases
Zillow	Bad data in home value model	$500 million loss

Data quality is not optional. It's survival.

The Data Quality Pyramid

```
        ┌───────────┐
        │  MODELS   │  ← What we want to build
        ├───────────┤
      ┌─┴───────────┴─┐
      │   ANALYTICS   │  ← Insights & reports
      ├───────────────┤
    ┌─┴───────────────┴─┐
    │    CLEAN DATA     │  ← Validated, typed
    ├───────────────────┤
  ┌─┴───────────────────┴─┐
  │    VALIDATED DATA     │  ← Schema-compliant
  ├───────────────────────┤
┌─┴───────────────────────┴─┐
│       RAW DATA            │  ← What we collected
└───────────────────────────┘
```

You can't skip layers. Each depends on the one below.

The Cost of Skipping Validation

The 1-10-100 Rule: It costs $1 to verify data at entry, $10 to fix it later, and $100 to recover from bad decisions made with bad data.

Where do problems get discovered?

Stage	Discovery Cost	Example
Data Entry	$1	Validation rejects bad input
Processing	$10	ETL pipeline fails
Analysis	$50	Analyst spots anomaly in report
Production	$100+	Model makes bad predictions
Business Impact	$1000+	Wrong decisions based on flawed data

Earlier is always cheaper.

Today's Mission

Transform messy raw data into clean, validated data.

Tools we'll learn:

- **Unix commands**: `head`, `tail`, `wc`, `file`, `sort`, `uniq`
- **jq**: JSON processing powerhouse
- **CSVkit**: CSV Swiss Army knife
- **JSON Schema**: Language-agnostic data contracts
- **Pydantic**: Pythonic data validation

Principle: Inspect before you trust. Validate before you use.

Part 2: Types of Data Problems

Know your enemy

A Taxonomy of Data Problems

```
┌──────────────────────────────────────────────────────────────┐
│                   DATA QUALITY DIMENSIONS                    │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│   COMPLETENESS  - Is all expected data present?              │
│   ACCURACY      - Is the data correct?                       │
│   CONSISTENCY   - Does data agree across sources?            │
│   VALIDITY      - Does data conform to rules?                │
│   UNIQUENESS    - Are there duplicates?                      │
│   TIMELINESS    - Is data up-to-date?                        │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Let's see examples of each...

Problem 1: Missing Values

The data simply isn't there.

```
title,year,rating,revenue
Inception,2010,8.8,292576195
Avatar,2009,7.9,2923706026
The Room,2003,3.9,
Tenet,,7.3,363656624
```

Types of missingness:

- **Empty string**: `""`
- **Null/None**: `null` in JSON
- **Sentinel value**: `"N/A"`, `"NULL"`, `-1`, `9999`
- **Missing key**: Key doesn't exist in JSON

Why it matters: ML models can't handle missing values directly.

Problem 2: Wrong Data Types

Data exists but in wrong format.

```
{
  "title": "Inception",
  "year": "2010",          // String, should be integer
  "rating": "8.8",         // String, should be float
  "runtime": "148 min",    // String with unit, should be integer
  "released": "16 Jul 2010" // String, should be date
}
```

Common type issues:

- Numbers stored as strings
- Dates in various string formats
- Booleans as "true"/"false"/"yes"/"no"/"1"/"0"
- Lists stored as comma-separated strings

Problem 3: Inconsistent Formats

Same concept, different representations.

```
# Date formats
2010-07-16
07/16/2010
16 Jul 2010
July 16, 2010

# Currency formats
$292,576,195
292576195
$292.5M
292,576,195 USD

# Boolean formats
true, True, TRUE, 1, yes, Yes, Y
```

Why it matters: Can't compare or aggregate inconsistent data.

Problem 4: Duplicates

Same record appears multiple times.

```
title,year,rating
Inception,2010,8.8
Avatar,2009,7.9
Inception,2010,8.8      ← Exact duplicate
The Matrix,1999,8.7
inception,2010,8.8      ← Case variation duplicate
Inception,2010,8.9      ← Near duplicate (different rating?)
```

Types of duplicates:

- **Exact**: Identical in every field
- **Partial**: Same key, different values (which is correct?)
- **Fuzzy**: Similar but not identical ("Spiderman" vs "Spider-Man")

Problem 5: Outliers and Anomalies

Values that are technically valid but suspicious.

```
title,year,rating,budget
Inception,2010,8.8,160000000
Avatar,2009,7.9,237000000
The Room,2003,3.9,6000000
Avengers,2012,8.0,-50000000     ← Negative budget?
Unknown,2025,9.9,999999999999   ← Future year, impossible rating
```

Questions to ask:

- Is this value within reasonable range?
- Is this value possible given business rules?
- Is this value consistent with other fields?

Problem 6: Encoding Issues

Text looks garbled or contains strange characters.

```
Expected: "Amelie"
Got:      "AmÃ©lie"      ← UTF-8 read as Latin-1

Expected: "Japanese text"
Got:      "æ¥æ¬èª"        ← Wrong encoding

Expected: "Zoe"
Got:      "Zo\xeb"        ← Raw bytes shown
```

Common encoding issues:

- UTF-8 vs Latin-1 (ISO-8859-1)
- Windows-1252 vs UTF-8
- BOM (Byte Order Mark) at file start

Problem 7: Schema Violations

Data structure doesn't match expectations.

```
// Expected schema
{"title": "string", "year": "integer", "genres": ["string"]}

// Actual data
{"title": "Inception", "year": 2010, "genres": ["Sci-Fi", "Action"]}  // OK
{"title": "Avatar", "year": "2009", "genres": "Action"}               // year is string, genres is string not array
{"Title": "Matrix", "Year": 1999}                                      // Wrong case, missing genres
{"title": null, "year": 2020, "genres": []}                           // Null title
```

Schema defines: Field names, types, required fields, constraints.

Summary: Data Problem Checklist

Problem	Question to Ask	Tool to Detect
Missing	Are there nulls/empty values?	`csvstat`, pandas
Types	Are numbers actually numbers?	`file`, schema validation
Format	Is date format consistent?	`grep`, regex
Duplicates	Are there repeated rows?	`sort`, `uniq`, `csvsql`
Outliers	Are values in valid range?	`csvstat`, histograms
Encoding	Is text readable?	`file`, `iconv`
Schema	Does structure match spec?	JSON Schema, Pydantic

Part 3: First Look at Your Data

Unix tools for initial inspection

Before You Do Anything: Look at the Data

Golden Rule: Never process data you haven't inspected.

```
# What kind of file is this?
file movies.csv

# How big is it?
ls -lh movies.csv
wc -l movies.csv

# What does it look like?
head movies.csv
tail movies.csv
```

These 5 commands should be muscle memory.

The `file` Command

Tells you what type of file you're dealing with.

```
$ file movies.csv
movies.csv: UTF-8 Unicode text, with CRLF line terminators

$ file movies.json
movies.json: JSON data

$ file data.xlsx
data.xlsx: Microsoft Excel 2007+

$ file mystery_file
mystery_file: gzip compressed data
```

Reveals:

- Text encoding (UTF-8, ASCII, ISO-8859-1)
- Line endings (LF vs CRLF)
- File format (CSV, JSON, binary)

The `wc` Command

Word count - but more useful for lines and characters.

```
$ wc movies.csv
  1001   5823  142567 movies.csv
  |      |      |
  |      |      +-- bytes
  |      +--------- words
  +---------------- lines

# Just line count (most common)
$ wc -l movies.csv
1001 movies.csv

# 1001 lines = 1 header + 1000 data rows
```

Quick sanity check: Expected 1000 movies? Check line count!

The `head` Command

See the first N lines of a file.

```
# First 10 lines (default)
$ head movies.csv
title,year,rating,revenue
Inception,2010,8.8,292576195
Avatar,2009,7.9,2923706026
...

# First 5 lines
$ head -n 5 movies.csv

# First 20 lines
$ head -20 movies.csv
```

Use case: Quickly see headers and sample data.

The `tail` Command

See the last N lines of a file.

```
# Last 10 lines
$ tail movies.csv

# Last 5 lines
$ tail -n 5 movies.csv

# Everything EXCEPT first line (skip header!)
$ tail -n +2 movies.csv
```

Use case: Check if file ends properly, skip headers.

Combining head and tail

See a slice of the file:

```
# Lines 100-110 (skip 99, take 11)
$ head -110 movies.csv | tail -11

# See header + specific row range
$ head -1 movies.csv && sed -n '500,510p' movies.csv
```

Practical example:

```
# File has 1 million rows, peek at middle
$ head -500000 huge.csv | tail -10
```

The `sort` Command

Sort lines alphabetically or numerically.

```
# Sort alphabetically
$ sort movies.csv

# Sort numerically on column 3 (rating)
$ sort -t',' -k3 -n movies.csv

# Sort in reverse (descending)
$ sort -t',' -k3 -nr movies.csv
```

Flags:

- `-t','` = field delimiter is comma
- `-k3` = sort by 3rd field
- `-n` = numeric sort
- `-r` = reverse

The `uniq` Command

Find or remove duplicate lines.

```
# Remove adjacent duplicates (MUST sort first!)
$ sort movies.csv | uniq

# Count occurrences of each line
$ sort movies.csv | uniq -c

# Show only duplicates
$ sort movies.csv | uniq -d

# Show only unique lines (appear once)
$ sort movies.csv | uniq -u
```

Important: `uniq` only detects *adjacent* duplicates. Always `sort` first!

Finding Duplicates: Practical Example

```
# How many duplicate titles?
$ cut -d',' -f1 movies.csv | sort | uniq -d
Inception
The Matrix
Spider-Man

# How many times does each duplicate appear?
$ cut -d',' -f1 movies.csv | sort | uniq -c | sort -rn | head
   3 Spider-Man
   2 The Matrix
   2 Inception
   1 Zodiac
   1 Zoolander
```

Found 3 duplicate titles!

The `cut` Command

Extract columns from delimited data.

```
# Get first column (titles)
$ cut -d',' -f1 movies.csv

# Get columns 1 and 3 (title and rating)
$ cut -d',' -f1,3 movies.csv

# Get columns 2 through 4
$ cut -d',' -f2-4 movies.csv
```

Flags:

- `-d','` = delimiter is comma
- `-f1` = first field
- `-f1,3` = fields 1 and 3
- `-f2-4` = fields 2 through 4

The `grep` Command

Search for patterns in text.

```
# Find rows containing "Inception"
$ grep "Inception" movies.csv

# Count matches
$ grep -c "N/A" movies.csv
47

# Show line numbers
$ grep -n "N/A" movies.csv

# Invert match (lines NOT containing)
$ grep -v "N/A" movies.csv

# Case insensitive
$ grep -i "matrix" movies.csv
```

Putting It Together: Initial Inspection Script

```bash
#!/bin/bash
FILE=$1

echo "=== File Info ==="
file "$FILE"
ls -lh "$FILE"

echo -e "\n=== Line Count ==="
wc -l "$FILE"

echo -e "\n=== First 5 Lines ==="
head -5 "$FILE"

echo -e "\n=== Last 5 Lines ==="
tail -5 "$FILE"

echo -e "\n=== Potential Issues ==="
echo "N/A values: $(grep -c 'N/A' "$FILE")"
echo "Empty fields: $(grep -c ',,' "$FILE")"
echo "Duplicate lines: $(sort "$FILE" | uniq -d | wc -l)"
```

Part 4: jq - JSON Processing

The Swiss Army knife for JSON

Why jq?

JSON is everywhere:

- API responses
- Configuration files
- Log files
- NoSQL databases

Problem: JSON is hard to read and process in shell.

```
# Raw JSON - unreadable mess
$ cat movies.json
{"Title":"Inception","Year":"2010","Rated":"PG-13","Released":"16 Jul 2010","Runtime":"148 min","Genre":"Action, Adventure, Sci-Fi"}
```

Solution: `jq` - a lightweight JSON processor.

The jq Mental Model

Think of jq as a pipeline: Data flows in, gets transformed, flows out. Each filter transforms the data for the next filter.

```
Input JSON  ⟶  Filter 1  ⟶  Filter 2  ⟶  Filter 3  ⟶  Output
    .          .movies       .[0]         .title       "Inception"
 (whole doc)  (get field)  (first elem)  (get title)
```

Key concepts:

- `.` = current data (identity)
- `|` = pipe to next filter
- `[]` = iterate over array
- `.field` = access object field

jq is like SQL for JSON - query and transform in one line.

jq Basics: Pretty Printing

```
# The identity filter: just pretty print
$ cat movie.json | jq .
{
  "Title": "Inception",
  "Year": "2010",
  "Rated": "PG-13",
  "Runtime": "148 min",
  "Genre": "Action, Adventure, Sci-Fi"
}
```

The `.` is the identity filter - it means "the whole input".

jq: Extracting Fields

```
# Get a single field
$ cat movie.json | jq '.Title'
"Inception"

# Get nested field
$ cat movie.json | jq '.Director.Name'
"Christopher Nolan"

# Get multiple fields
$ cat movie.json | jq '.Title, .Year'
"Inception"
"2010"
```

Syntax: `.fieldname` extracts that field.

jq: Working with Arrays

```
// movies.json - array of movies
[
  {"Title": "Inception", "Year": "2010"},
  {"Title": "Avatar", "Year": "2009"},
  {"Title": "The Matrix", "Year": "1999"}
]
```

```
# Get first element
$ cat movies.json | jq '.[0]'
{"Title": "Inception", "Year": "2010"}

# Get all titles
$ cat movies.json | jq '.[].Title'
"Inception"
"Avatar"
"The Matrix"

# Get length of array
$ cat movies.json | jq 'length'
3
```

jq: The Array Iterator `[]`

```
# .[] iterates over array elements
$ cat movies.json | jq '.[]'
{"Title": "Inception", "Year": "2010"}
{"Title": "Avatar", "Year": "2009"}
{"Title": "The Matrix", "Year": "1999"}

# Chain with field extraction
$ cat movies.json | jq '.[].Title'
"Inception"
"Avatar"
"The Matrix"

# Same as:
$ cat movies.json | jq '.[] | .Title'
```

The pipe `|` passes output to next filter.

jq: Building New Objects

```
# Create new object structure
$ cat movies.json | jq '.[] | {name: .Title, year: .Year}'
{"name": "Inception", "year": "2010"}
{"name": "Avatar", "year": "2009"}
{"name": "The Matrix", "year": "1999"}

# Wrap results in array
$ cat movies.json | jq '[.[] | {name: .Title, year: .Year}]'
[
  {"name": "Inception", "year": "2010"},
  {"name": "Avatar", "year": "2009"},
  {"name": "The Matrix", "year": "1999"}
]
```

jq: Filtering with `select()`

```
# Filter movies from 2010 or later
$ cat movies.json | jq '.[] | select(.Year >= "2010")'
{"Title": "Inception", "Year": "2010"}

# Filter by string match
$ cat movies.json | jq '.[] | select(.Title == "Avatar")'
{"Title": "Avatar", "Year": "2009"}

# Filter by pattern (contains)
$ cat movies.json | jq '.[] | select(.Title | contains("The"))'
{"Title": "The Matrix", "Year": "1999"}
```

jq: Type Conversion

Remember: API data often has numbers as strings!

```
# Convert string to number
$ echo '{"year": "2010"}' | jq '.year | tonumber'
2010

# Now we can do numeric comparisons
$ cat movies.json | jq '.[] | select((.Year | tonumber) >= 2005)'

# Convert number to string
$ echo '{"count": 42}' | jq '.count | tostring'
"42"
```

jq: Handling Missing Data

```
# Optional field access (no error if missing)
$ echo '{"title": "X"}' | jq '.rating'
null

# Provide default value
$ echo '{"title": "X"}' | jq '.rating // "N/A"'
"N/A"

# Check if field exists
$ echo '{"title": "X"}' | jq 'has("rating")'
false

# Filter out nulls
$ cat movies.json | jq '.[] | select(.Rating != null)'
```

jq: Aggregation Functions

```
# Count elements
$ cat movies.json | jq 'length'
100

# Get unique values
$ cat movies.json | jq '[.[].Genre] | unique'
["Action", "Comedy", "Drama", "Sci-Fi"]

# Min and max
$ cat movies.json | jq '[.[].Year | tonumber] | min'
1999
$ cat movies.json | jq '[.[].Year | tonumber] | max'
2023

# Sum and average
$ cat movies.json | jq '[.[].Rating | tonumber] | add'
725.5
$ cat movies.json | jq '[.[].Rating | tonumber] | add / length'
7.255
```

jq: Sorting

```
# Sort array of objects by field
$ cat movies.json | jq 'sort_by(.Year)'

# Sort descending (reverse)
$ cat movies.json | jq 'sort_by(.Year) | reverse'

# Sort by numeric field
$ cat movies.json | jq 'sort_by(.Rating | tonumber) | reverse'

# Get top 5 rated movies
$ cat movies.json | jq 'sort_by(.Rating | tonumber) | reverse | .[0:5]'
```

jq: Grouping

```
# Group movies by year
$ cat movies.json | jq 'group_by(.Year)'
[
  [{"Title": "The Matrix", "Year": "1999"}],
  [{"Title": "Avatar", "Year": "2009"}],
  [{"Title": "Inception", "Year": "2010"}, {"Title": "Toy Story 3", "Year": "2010"}]
]

# Count movies per year
$ cat movies.json | jq 'group_by(.Year) | map({year: .[0].Year, count: length})'
[
  {"year": "1999", "count": 1},
  {"year": "2009", "count": 1},
  {"year": "2010", "count": 2}
]
```

jq: Raw Output Mode

```
# Default: outputs JSON strings with quotes
$ cat movies.json | jq '.[].Title'
"Inception"
"Avatar"

# Raw mode: no quotes (useful for scripting)
$ cat movies.json | jq -r '.[].Title'
Inception
Avatar

# Create CSV output
$ cat movies.json | jq -r '.[] | [.Title, .Year, .Rating] | @csv'
"Inception","2010","8.8"
"Avatar","2009","7.9"

# Create TSV output
$ cat movies.json | jq -r '.[] | [.Title, .Year] | @tsv'
Inception	2010
Avatar	2009
```

jq: Practical Data Validation Examples

```
# Find movies with missing ratings
$ cat movies.json | jq '[.[] | select(.Rating == null or .Rating == "N/A")] | length'
47

# Find movies with invalid years
$ cat movies.json | jq '.[] | select((.Year | tonumber) > 2024 or (.Year | tonumber) < 1900)'

# List all unique values in a field (check for variants)
$ cat movies.json | jq '[.[].Rated] | unique'
["G", "PG", "PG-13", "R", "Not Rated", "N/A", null]

# Find duplicate titles
$ cat movies.json | jq 'group_by(.Title) | map(select(length > 1)) | .[].Title'
```

jq Cheat Sheet - Basics

<table>
<tr><th>Task</th><th>Command</th></tr>
<tr><td>Pretty print</td><td>jq .</td></tr>
<tr><td>Get field</td><td>jq '.fieldname'</td></tr>
<tr><td>Get nested</td><td>jq '.a.b.c'</td></tr>
<tr><td>Array element</td><td>jq '.[0]'</td></tr>
<tr><td>All elements</td><td>jq '.[]'</td></tr>
<tr><td>Filter</td><td>jq '.[] | select(.x > 5)'</td></tr>
</table>

jq Cheat Sheet - Advanced

Task	Command
Build object	`jq '{a: .x, b: .y}'`
Count	`jq 'length'`
Sort	`jq 'sort_by(.field)'`
Unique	`jq 'unique'`
Raw strings	`jq -r`

Part 5: CSVkit

The CSV Swiss Army Knife

Why CSVkit?

CSV looks simple but hides complexity:

- Quoted fields with commas inside
- Multiline values
- Different delimiters
- Inconsistent escaping

CSVkit: A suite of command-line tools for CSV files.

```
# Installation
pip install csvkit
```

Tools we'll cover:

`csvlook`, `csvstat`, `csvcut`, `csvgrep`, `csvsort`, `csvjson`, `csvsql`

csvlook: Pretty Print CSV

Makes CSV readable in terminal.

```
$ csvlook movies.csv
| title      | year | rating | revenue     |
| ---------- | ---- | ------ | ----------- |
| Inception  | 2010 |    8.8 | 292576195   |
| Avatar     | 2009 |    7.9 | 2923706026  |
| The Matrix | 1999 |    8.7 | 463517383   |
| The Room   | 2003 |    3.9 |             |
```

Compare to raw:

```
title,year,rating,revenue
Inception,2010,8.8,292576195
Avatar,2009,7.9,2923706026
```

csvstat: Data Profiling

Get statistics for every column automatically!

```
$ csvstat movies.csv
  1. "title"
        Type of data:          Text
        Contains null values:  False
        Unique values:         987
        Longest value:         45 characters
        Most common values:    Spider-Man (3x)
                               The Matrix (2x)

  2. "year"
        Type of data:          Number
        Contains null values:  True (13 nulls)
        Smallest value:        1920
        Largest value:         2024
        Mean:                  2005.3
```

csvstat: Specific Columns

```
# Stats for just one column
$ csvstat -c rating movies.csv
  3. "rating"
        Type of data:          Number
        Contains null values:  True (108 nulls)
        Smallest value:        1.2
        Largest value:         9.3
        Mean:                  6.84
        Median:                7.1
        StDev:                 1.23

# Stats for multiple columns
$ csvstat -c year,rating movies.csv

# Just show counts
$ csvstat --count movies.csv
1000
```

csvcut: Select Columns

```
# Select by column name
$ csvcut -c title,year movies.csv
title,year
Inception,2010
Avatar,2009

# Select by column number
$ csvcut -c 1,3 movies.csv

# Exclude columns
$ csvcut -C revenue movies.csv

# List column names
$ csvcut -n movies.csv
  1: title
  2: year
  3: rating
  4: revenue
```

csvgrep: Filter Rows

```
# Filter by exact match
$ csvgrep -c year -m "2010" movies.csv

# Filter by regex pattern
$ csvgrep -c title -r "^The" movies.csv    # Starts with "The"

# Filter by inverse (NOT matching)
$ csvgrep -c rating -m "N/A" -i movies.csv  # Exclude N/A

# Filter for empty values
$ csvgrep -c revenue -r "^$" movies.csv     # Empty revenue
```

csvsort: Sort Data

```
# Sort by column
$ csvsort -c year movies.csv

# Sort descending
$ csvsort -c rating -r movies.csv

# Sort by multiple columns
$ csvsort -c year,rating movies.csv

# Numeric sort happens automatically for number columns!
```

csvjson: Convert to JSON

```
# CSV to JSON array
$ csvjson movies.csv
[
  {"title": "Inception", "year": 2010, "rating": 8.8},
  {"title": "Avatar", "year": 2009, "rating": 7.9}
]

# Indented output
$ csvjson -i 2 movies.csv

# JSON to CSV (reverse)
$ cat movies.json | in2csv -f json > movies.csv
```

Great for converting between formats!

csvsql: Query CSV with SQL!

Yes, you can run SQL on CSV files.

```
# Run SQL query
$ csvsql --query "SELECT title, rating FROM movies WHERE year > 2010" movies.csv

# Find duplicates
$ csvsql --query "SELECT title, COUNT(*) as cnt
                  FROM movies
                  GROUP BY title
                  HAVING cnt > 1" movies.csv

# Join two CSV files
$ csvsql --query "SELECT m.title, g.genre
                  FROM movies m
                  JOIN genres g ON m.id = g.movie_id" movies.csv genres.csv
```

csvsql: Data Validation Queries

```
# Find rows with missing values
$ csvsql --query "SELECT * FROM movies WHERE rating IS NULL" movies.csv

# Find out-of-range values
$ csvsql --query "SELECT * FROM movies WHERE year < 1900 OR year > 2025" movies.csv

# Find suspiciously high values
$ csvsql --query "SELECT * FROM movies WHERE revenue > 5000000000" movies.csv
```

csvclean: Fix Common Issues

```
# Check for problems (dry run)
$ csvclean -n movies.csv
1 error found:
Line 47: Expected 4 columns, found 5

# Fix and create cleaned file
$ csvclean movies.csv
# Creates movies_out.csv (cleaned) and movies_err.csv (errors)

# Common fixes:
# - Removes rows with wrong column count
# - Normalizes quoting
# - Reports line numbers of errors
```

CSVkit Pipeline Example

```
# Full data validation pipeline
$ cat movies.csv \
  | csvclean -n 2>&1 | head -5           # Check for structural issues

$ csvstat -c year,rating movies.csv      # Profile key columns

$ csvgrep -c rating -m "N/A" movies.csv \
  | csvcut -c title,year                  # Find movies with N/A rating

$ csvsql --query \
  "SELECT year, COUNT(*) as count, AVG(rating) as avg_rating
   FROM movies
   GROUP BY year
   ORDER BY year DESC" movies.csv         # Aggregate stats
```

CSVkit Cheat Sheet - Core Tools

Tool	Purpose	Example
`csvlook`	Pretty print	`csvlook data.csv`
`csvstat`	Statistics	`csvstat -c column data.csv`
`csvcut`	Select columns	`csvcut -c col1,col2 data.csv`
`csvgrep`	Filter rows	`csvgrep -c col -m "value"`
`csvsort`	Sort	`csvsort -c col -r data.csv`

CSVkit Cheat Sheet - Advanced Tools

Tool	Purpose	Example
`csvjson`	To JSON	`csvjson data.csv`
`csvsql`	SQL queries	`csvsql --query "..."`
`csvclean`	Fix issues	`csvclean data.csv`
`csvjoin`	Join files	`csvjoin -c id a.csv b.csv`
`csvstack`	Concatenate	`csvstack a.csv b.csv`

Part 6: Data Profiling

Understanding your data before using it

What is Data Profiling?

Data profiling = Analyzing data to understand its structure, content, and quality.

```
┌──────────────────────────────────────────────────────────┐
│                 DATA PROFILING QUESTIONS                 │
├──────────────────────────────────────────────────────────┤
│   STRUCTURE:     How many rows? Columns? What types?     │
│   COMPLETENESS:  How many nulls per column?              │
│   UNIQUENESS:    How many distinct values? Duplicates?   │
│   DISTRIBUTION:  Min, max, mean, median? Outliers?       │
│   PATTERNS:      What formats are used? Any anomalies?   │
└──────────────────────────────────────────────────────────┘
```

Profiling Step 1: Basic Shape

```
# How many rows and columns?
$ head -1 movies.csv | tr ',' '\n' | wc -l      # columns
5

$ wc -l movies.csv                              # rows (including header)
1001

# Or with csvstat
$ csvstat --count movies.csv
1000
```

First sanity check: Does shape match expectations?

Profiling Step 2: Column Types

```
$ csvstat movies.csv 2>&1 | grep "Type of data"
        Type of data:          Text
        Type of data:          Number
        Type of data:          Number
        Type of data:          Number
        Type of data:          Text

# Expected: title(text), year(int), rating(float), revenue(int), genre(text)
# Actual: Matches! But let's verify...
```

Profiling Step 3: Null Analysis

```
# Count nulls per column
$ csvstat movies.csv 2>&1 | grep -A1 "Contains null"
        Contains null values:  False
--
        Contains null values:  True (13 nulls)
--
        Contains null values:  True (108 nulls)
--
        Contains null values:  True (366 nulls)
```

Results:

- Title: 0 nulls (good!)
- Year: 13 nulls (1.3%)
- Rating: 108 nulls (10.8%)
- Revenue: 366 nulls (36.6%) - **problem!**

Profiling Step 4: Unique Values

```
# How many distinct values?
$ csvstat movies.csv 2>&1 | grep "Unique values"
        Unique values:         987      # title - expect 1000, so ~13 duplicates
        Unique values:         85       # year - reasonable range
        Unique values:         78       # rating - 1.0 to 10.0 scale
        Unique values:         634      # revenue - 634 non-null values

# Most common values (find duplicates, common patterns)
$ csvstat movies.csv 2>&1 | grep -A5 "Most common values"
```

Profiling Step 5: Value Ranges

```
$ csvstat -c year movies.csv
        Smallest value:       1920
        Largest value:        2024
        Mean:                 2005.3
        Median:               2010
        StDev:                15.2

# Check for suspicious outliers
# 1920 seems old - is it valid?
# 2024 is current year - any future years?
```

```
# Find extremes
$ csvsort -c year movies.csv | head -5      # oldest
$ csvsort -c year -r movies.csv | head -5   # newest
```

Profiling Step 6: Pattern Detection

```
# What values does 'rating' column have?
$ csvcut -c rating movies.csv | sort | uniq -c | sort -rn | head
    892 (valid numbers 1.0-10.0)
     47 N/A
     38
     23 Not Rated

# Aha! Three types of "missing":
# 1. Empty string
# 2. "N/A" string
# 3. "Not Rated" string
```

This is why automated profiling misses things!

Profiling Summary: Movies Dataset

Column	Type	Nulls	Unique	Issues
title	Text	0	987	13 duplicates
year	Int	13	85	1920-2024 range
rating	Float	108	78	"N/A", empty, "Not Rated"
revenue	Int	366	634	36% missing!
genre	Text	0	23	Multi-value ("Action, Drama")

Key findings:

1. Revenue is missing for 1/3 of movies
2. Rating has multiple representations of "missing"
3. There are 13 duplicate titles
4. Genre contains multiple values in one field

Part 7: Schema Validation

Contracts for your data

What is a Schema?

Schema = A formal description of expected data structure.

```
┌──────────────────────────────────────────────────────────────┐
│                       SCHEMA DEFINES                         │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│  FIELD NAMES:      What columns/keys should exist?           │
│  DATA TYPES:       String, integer, float, boolean, array?   │
│  CONSTRAINTS:      Required? Min/max? Pattern? Enum?         │
│  RELATIONSHIPS:    References to other data?                 │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Analogy: A schema is like a contract between data producer and consumer.

Schema: The Blueprint Analogy

Think of a schema like a building blueprint: Before construction begins, everyone agrees on what the building should look like. The blueprint defines rooms, dimensions, materials - and the building must match.

Without blueprint (schema):

- Builder guesses what's needed
- Inspector can't verify if it's correct
- Different workers make inconsistent decisions
- Problems discovered when building collapses

With blueprint (schema):

- Clear expectations documented upfront
- Automatic verification at each step
- Everyone builds the same thing

Why Schemas Matter

Without schema:

```
# What is this?
data = {"yr": 2010, "rt": "8.8", "ttl": "Inception"}
```

With schema:

```
# Clear expectations
schema = {
    "title": {"type": "string", "required": True},
```

Schemas enable:

- Automatic validation
- Documentation
- Code generation
- Early error detection

JSON Schema: The Standard

JSON Schema is a vocabulary for validating JSON data.

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "title": {
      "type": "string",
      "minLength": 1
    },
    "year": {
      "type": "integer",
      "minimum": 1880,
      "maximum": 2030
    },
    "rating": {
      "type": "number",
      "minimum": 0,
      "maximum": 10
    }
  },
  "required": ["title", "year"]
```

JSON Schema: Type Keywords

Keyword	Valid Values
`"type": "string"`	`"hello"`, `""`
`"type": "integer"`	`42`, `-1`, `0`
`"type": "number"`	`3.14`, `42`, `-1.5`
`"type": "boolean"`	`true`, `false`
`"type": "null"`	`null`
`"type": "array"`	`[1, 2, 3]`, `[]`
`"type": "object"`	`{"a": 1}`, `{}`

Multiple types:

```
{"type": ["string", "null"]}   // String or null
```

JSON Schema: String Constraints

```
{
  "type": "string",
  "minLength": 1,                // At least 1 character
  "maxLength": 100,              // At most 100 characters
  "pattern": "^[A-Z].*$",        // Must start with uppercase
  "format": "email"              // Must be valid email
}
```

Common formats:

- `"email"` - Email address
- `"date"` - ISO 8601 date (2010-07-16)
- `"date-time"` - ISO 8601 datetime
- `"uri"` - Valid URI
- `"uuid"` - UUID format

JSON Schema: Number Constraints

```
{
  "type": "number",
  "minimum": 0,                // ≥ 0
  "maximum": 10,               // ≤ 10
  "exclusiveMinimum": 0,       // > 0
  "exclusiveMaximum": 10,      // < 10
  "multipleOf": 0.1            // Must be multiple of 0.1
}
```

Example for rating:

```
{
  "type": "number",
  "minimum": 0,
  "maximum": 10,
  "multipleOf": 0.1
}
```

JSON Schema: Arrays

```
{
  "type": "array",
  "items": {
    "type": "string"           // All items must be strings
  },
  "minItems": 1,               // At least 1 item
  "maxItems": 10,              // At most 10 items
  "uniqueItems": true          // No duplicates
}
```

Example for genres:

```
{
  "genres": {
    "type": "array",
    "items": {"type": "string"},
    "minItems": 1
  }
}
```

JSON Schema: Enums

Restrict to specific values:

```json
{
  "rated": {
    "type": "string",
    "enum": ["G", "PG", "PG-13", "R", "NC-17", "Not Rated"]
  }
}
```

Validation result:

- `"PG-13"` - Valid
- `"PG13"` - Invalid (not in enum)
- `"M"` - Invalid (not in enum)

JSON Schema: Required Fields

```
{
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "year": {"type": "integer"},
    "rating": {"type": "number"},
    "revenue": {"type": "integer"}
  },
  "required": ["title", "year"]    // Only title and year are required
}
```

Validation:

- `{"title": "X", "year": 2010}` - Valid (rating, revenue optional)
- `{"title": "X"}` - Invalid (missing required field: year)

Complete Movie Schema Example

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "title": {"type": "string", "minLength": 1},
    "year": {"type": "integer", "minimum": 1880, "maximum": 2030},
    "rating": {"type": ["number", "null"], "minimum": 0, "maximum": 10},
    "revenue": {"type": ["integer", "null"], "minimum": 0},
    "genres": {
      "type": "array",
      "items": {"type": "string"},
      "minItems": 1
    },
    "rated": {
      "type": "string",
      "enum": ["G", "PG", "PG-13", "R", "NC-17", "Not Rated"]
    }
  },
  "required": ["title", "year", "genres"]
}
```

Validating with Python

```python
import json
from jsonschema import validate, ValidationError

schema = {
    "type": "object",
    "properties": {
        "title": {"type": "string"},
        "year": {"type": "integer", "minimum": 1880}
    },
    "required": ["title", "year"]
}

movie = {"title": "Inception", "year": 2010}

try:
    validate(instance=movie, schema=schema)
    print("Valid!")
except ValidationError as e:
    print(f"Invalid: {e.message}")
```

Schema-First Development

Traditional approach:

1. Collect data
2. Write code to process it
3. Discover problems in production

Schema-first approach:

1. Define schema (contract)
2. Validate data against schema on ingestion
3. Reject invalid data early
4. Process only valid data

Part 8: Pydantic

Pythonic data validation

Why Pydantic?

JSON Schema limitations:

- Separate from your Python code
- No IDE autocompletion
- Manual validation calls
- Verbose error handling

Pydantic advantages:

- Uses Python type hints (you already know this!)
- Automatic validation on object creation
- IDE support (autocomplete, type checking)
- Clear, readable error messages
- Used by FastAPI, LangChain, and many modern libraries

Pydantic: Basic Model

```python
from pydantic import BaseModel

class Movie(BaseModel):
    title: str
    year: int
    rating: float

# Valid data - works!
movie = Movie(title="Inception", year=2010, rating=8.8)
print(movie.title)  # "Inception"
print(movie.year)   # 2010 (as int, not string!)
```

Key insight: Just define a class with type hints. Pydantic does the rest.

Pydantic: The Bouncer Analogy

Think of Pydantic like a nightclub bouncer: The bouncer checks everyone at the door. If you don't meet the requirements (dress code, age, etc.), you don't get in. Once inside, everyone is guaranteed to meet the standards.

```python
class Movie(BaseModel):  # ← The bouncer's checklist
    title: str           # Must have a name
    year: int            # Must be a valid year (number)
    rating: float        # Must have a rating (number)

# The bouncer checks at the door (object creation)
movie = Movie(**raw_data)  # ← Validation happens HERE

# Once past the bouncer, you're guaranteed valid
print(movie.year + 1)  # Safe - year is definitely an int
```

No more "is this a string or int?" questions inside your code.

Pydantic: Automatic Type Coercion

```python
# Pydantic converts types automatically when possible
movie = Movie(title="Inception", year="2010", rating="8.8")
print(movie.year)    # 2010 (converted from string to int)
print(movie.rating)  # 8.8 (converted from string to float)

# But invalid conversions fail
movie = Movie(title="Inception", year="not a year", rating=8.8)
# ValidationError: Input should be a valid integer
```

Principle: Be strict about structure, flexible about representation.

Pydantic: Validation Errors

```python
from pydantic import ValidationError

try:
    movie = Movie(title="", year=2010, rating=8.8)
except ValidationError as e:
    print(e)
```

```
1 validation error for Movie
title
  String should have at least 1 character [type=string_too_short]
```

Errors are clear: Field name, what's wrong, and why.

Pydantic: Field Constraints

```python
from pydantic import BaseModel, Field

class Movie(BaseModel):
    title: str = Field(min_length=1)
    year: int = Field(ge=1880, le=2030)  # ge = greater or equal
    rating: float = Field(ge=0, le=10)
    revenue: int | None = None  # Optional field
```

```python
Movie(title="X", year=1850, rating=8.0)
# ValidationError: year - Input should be ≥ 1880
```

Pydantic: Optional and Default Values

```python
from pydantic import BaseModel
from typing import Optional

class Movie(BaseModel):
    title: str
    year: int
    rating: Optional[float] = None      # Can be None
    genres: list[str] = []              # Default empty list
    is_released: bool = True            # Default value
```

```python
movie = Movie(title="Tenet", year=2020)
print(movie.rating)      # None
print(movie.genres)      # []
print(movie.is_released) # True
```

Pydantic vs JSON Schema

Aspect	JSON Schema	Pydantic
Language	JSON (separate file)	Python (in your code)
Type hints	No	Yes
IDE support	Limited	Full autocomplete
Validation	Manual call	Automatic on create
Error messages	Technical	Human-readable
Learning curve	New syntax	Just Python

Recommendation: Use Pydantic for Python projects, JSON Schema for APIs/cross-language.

Pydantic: The Mental Model

```
┌────────────────────────────────────────────────────────────┐
│                    PYDANTIC WORKFLOW                       │
├────────────────────────────────────────────────────────────┤
│                                                            │
│  1. DEFINE     class Movie(BaseModel): ...                 │
│                                                            │
│  2. CREATE     movie = Movie(**raw_data)                   │
│                      |                                     │
│                      v                                     │
│             [Validation happens HERE]                      │
│                      |                                     │
│            Valid?  / \  Invalid?                           │
│                   /   \                                    │
│  3. USE      movie.title   raise ValidationError           │
│                                                            │
└────────────────────────────────────────────────────────────┘
```

Pydantic: Practical Example

```python
from pydantic import BaseModel, Field
from typing import Optional

class MovieFromAPI(BaseModel):
    """Validates movie data from OMDB API."""
    Title: str = Field(min_length=1)
    Year: str  # API returns string, we'll convert later
    imdbRating: Optional[str] = None
    BoxOffice: Optional[str] = None

# Parse API response - validation happens automatically
raw = {"Title": "Inception", "Year": "2010", "imdbRating": "8.8"}
movie = MovieFromAPI(**raw)  # Works!

raw_bad = {"Title": "", "Year": "2010"}
movie = MovieFromAPI(**raw_bad)  # ValidationError!
```

What We'll Cover in Lab

Pydantic deep dive:

- Nested models (Movie with Director, Actors)
- Custom validators (`@validator` decorator)
- Parsing JSON files with Pydantic
- Model serialization (`.model_dump()`, `.model_dump_json()`)
- Strict mode vs coercion mode

The lab is where you'll get hands-on practice!

Part 9: Encoding & Edge Cases

When text isn't just text

The Encoding Problem

Computers store text as numbers. But which numbers?

```
Character 'A' = 65 (ASCII)
Character 'e' with accent = ??? (depends on encoding!)
```

Encoding = The mapping between characters and bytes.

```
┌──────────────────────────────────────────────────────────────────┐
│                        COMMON ENCODINGS                          │
├──────────────────────────────────────────────────────────────────┤
│   ASCII        - 128 characters (English only)                   │
│   Latin-1      - 256 characters (Western European)               │
│   UTF-8        - 1,112,064 characters (everything!)              │
│   UTF-16       - Same characters, different byte format          │
│   Windows-1252 - Microsoft's Latin-1 variant                     │
└──────────────────────────────────────────────────────────────────┘
```

UTF-8: The Modern Standard

UTF-8 is the dominant encoding for the web and modern systems.

Why UTF-8?

- Backwards compatible with ASCII
- Supports all languages
- Variable length (1-4 bytes per character)
- Self-synchronizing

```
# Check file encoding
$ file movies.csv
movies.csv: UTF-8 Unicode text

$ file old_data.csv
old_data.csv: ISO-8859-1 text
```

Encoding Problems in Practice

What you expect:

```
Amelie (with accent)
Crouching Tiger, Hidden Dragon (Chinese title)
```

What you get:

```
AmÃ©lie                    ← UTF-8 decoded as Latin-1
Crouching Tiger (????????)   ← Wrong encoding
```

Common scenarios:

1. File saved in one encoding, read in another
2. Copy-paste from web with different encoding
3. Database with mixed encodings
4. Legacy systems using old encodings

Detecting Encoding

```
# The file command guesses encoding
$ file -i movies.csv
movies.csv: text/plain; charset=utf-8

# For more accuracy, use chardet (Python)
$ pip install chardet
$ chardetect movies.csv
movies.csv: utf-8 with confidence 0.99

# Or with Python
$ python -c "import chardet; print(chardet.detect(open('movies.csv','rb').read()))"
{'encoding': 'utf-8', 'confidence': 0.99}
```

Converting Encodings

```
# Convert from Latin-1 to UTF-8
$ iconv -f ISO-8859-1 -t UTF-8 old_file.csv > new_file.csv

# Convert from Windows-1252 to UTF-8
$ iconv -f WINDOWS-1252 -t UTF-8 windows_file.csv > utf8_file.csv

# List available encodings
$ iconv -l
```

Python approach:

```python
# Read with specific encoding
with open('file.csv', encoding='latin-1') as f:
    content = f.read()

# Write as UTF-8
with open('file_utf8.csv', 'w', encoding='utf-8') as f:
    f.write(content)
```

CSV Edge Cases: Quoting

What if your data contains commas?

```
title,year,description
Inception,2010,A mind-bending, complex thriller    ← WRONG! Extra column
"Inception",2010,"A mind-bending, complex thriller" ← Correct: quoted
```

What if your data contains quotes?

```
title,year,tagline
Say "Hello",2020,A movie about "greetings"    ← WRONG!
"Say ""Hello""",2020,"A movie about ""greetings""" ← Correct: escaped
```

Rule: Fields with commas, quotes, or newlines must be quoted.

CSV Edge Cases: Line Endings

Different systems use different line endings:

System	Line Ending	Bytes
Unix/Linux/Mac	LF	`\n` (0x0A)
Windows	CRLF	`\r\n` (0x0D 0x0A)
Old Mac	CR	`\r` (0x0D)

Problems occur when mixing:

```
# Detect line endings
$ file data.csv
data.csv: ASCII text, with CRLF line terminators

# Convert Windows to Unix
$ sed -i 's/\r$//' data.csv
# Or
$ dos2unix data.csv
```

CSV Edge Cases: Multiline Values

Values can contain newlines (if quoted):

```
title,year,plot
"Inception",2010,"A thief who steals corporate secrets through dream-sharing
technology is given the inverse task of planting an idea into the mind
of a C.E.O."
"Avatar",2009,"A paraplegic Marine..."
```

This is valid CSV! But many simple parsers break.

Solution: Use proper CSV parsers (pandas, csvkit), not line-by-line reading.

CSV Edge Cases: Empty vs Null

What does this mean?

```
title,year,rating
Inception,2010,8.8
Avatar,2009,
The Room,2003,""
```

Row	rating value	Interpretation
1	`8.8`	Rating is 8.8
2	(nothing)	Rating is null/missing
3	`""`	Rating is empty string

Is empty string the same as null? Depends on your interpretation!

Handling Edge Cases: Best Practices

1. Always specify encoding explicitly:

```
pd.read_csv('file.csv', encoding='utf-8')
```

2. Use proper CSV parsers:

```
# Good
import csv
with open('file.csv') as f:
    reader = csv.reader(f)

# Bad
with open('file.csv') as f:
    for line in f:
        fields = line.split(',')  # Breaks on quoted commas!
```

3. Validate after reading:

```
assert df['year'].dtype == 'int64', "Year should be integer"
```

Part 10: Validation Principles

Best practices for data quality

Principle 1: Validate at the Boundary

Check data when it enters your system, not later.

```
┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│  External   │ →  │ Validation  │ →  │    Your     │
│  Data       │    │ Layer       │    │    System   │
└─────────────┘    └─────────────┘    └─────────────┘
                         |
                         v
                  [Invalid data rejected here]
```

Why?

- Invalid data doesn't spread through your system
- Easier to debug (you know exactly where it failed)
- Clear separation of concerns

Principle 2: Fail Fast

Stop immediately when you find invalid data.

```
# Bad: Continue and hope for the best
for movie in movies:
    try:
        process(movie)
    except:
        pass  # Silent failure!

# Good: Fail fast and loud
for movie in movies:
    validate(movie)  # Raises exception if invalid
    process(movie)
```

Benefits:

- Find problems early
- Don't waste time processing bad data
- Easier debugging

Principle 3: Be Explicit About Missing Data

Don't guess. Document and handle explicitly.

```python
# Bad: Implicit handling
rating = movie.get('rating', 0)  # Is 0 a valid rating or missing?

# Good: Explicit handling
rating = movie.get('rating')
if rating is None:
    raise ValidationError("Rating is required")
# Or
if rating is None:
    rating = DEFAULT_RATING  # Explicitly documented default
```

Principle 4: Validate Types AND Values

Type checking isn't enough.

```python
# Type is correct (integer), but value is invalid
year = -500     # Negative year
year = 9999     # Far future
year = 1066     # Before cinema existed

# Need both type AND range validation
def validate_year(year):
    if not isinstance(year, int):
        raise TypeError("Year must be integer")
    if year < 1880 or year > 2030:
        raise ValueError(f"Year {year} out of valid range")
```

Principle 5: Log Validation Failures

Keep records of what failed and why.

```python
import logging

def validate_movies(movies):
    valid = []
    for i, movie in enumerate(movies):
        try:
            validate(movie)
            valid.append(movie)
        except ValidationError as e:
            logging.warning(f"Row {i}: {e.message} - {movie}")
```

Why?

- Understand data quality trends
- Debug upstream issues
- Audit trail

Principle 6: Separate Validation from Cleaning

Two different operations:

Validation	Cleaning
Checks if data is valid	Fixes invalid data
Returns true/false	Modifies data
Should not modify	Requires decisions
Objective	Subjective

```python
# Validation: Does it pass?
def is_valid_year(year):
    return isinstance(year, int) and 1880 ⩽ year ⩽ 2030

# Cleaning: Make it pass
def clean_year(year_str):
    return int(year_str.strip())
```

Principle 7: Test Your Validation

Validation code needs tests too!

```
def test_year_validation():
    # Valid cases
    assert validate_year(2010) == True
    assert validate_year(1880) == True  # Boundary
    assert validate_year(2030) == True  # Boundary

    # Invalid cases
    assert validate_year(1879) == False  # Just below
    assert validate_year(2031) == False  # Just above
    assert validate_year("2010") == False  # Wrong type
    assert validate_year(None) == False   # Null
```

Edge cases are where bugs hide!

Common Validation Mistakes

Mistakes that let bad data slip through:

Mistake	Example	Better Approach
Only checking type	`isinstance(x, int)`	Also check range: `0 < x < 1000`
Trusting "not None"	`if value:`	Empty string `""` is falsy but not None
Case sensitivity	`if status == "active"`	`if status.lower() == "active"`
Whitespace	`if name == "John"`	`if name.strip() == "John"`
Encoding	Reading UTF-8 as ASCII	Always specify encoding
Off-by-one	`year < 2024`	Should it be `<= 2024`?

Rule of thumb: If something CAN go wrong, it WILL. Validate defensively.

Part 11: Building a Validation Pipeline

Putting it all together

The Validation Pipeline

```
┌──────────┐   ┌──────────┐   ┌──────────┐   ┌──────────┐
│ Ingest   │ → │ Inspect  │ → │ Validate │ → │ Clean    │
│ Raw Data │   │ Profile  │   │ Schema   │   │ Transform│
└──────────┘   └──────────┘   └──────────┘   └──────────┘
                                   |
                                   v
                          [Reject/Quarantine
                           Invalid Records]
```

Four stages:

1. **Ingest**: Load raw data
2. **Inspect**: Profile and understand
3. **Validate**: Check against rules
4. **Clean**: Fix and transform

Stage 1: Ingest

```bash
# Download or receive data
curl -o movies_raw.json "$API_URL"

# Check what we got
file movies_raw.json
wc -l movies_raw.json
head movies_raw.json | jq .
```

```python
# Load with explicit encoding
import json
with open('movies_raw.json', encoding='utf-8') as f:
    movies = json.load(f)
print(f"Loaded {len(movies)} movies")
```

Stage 2: Inspect and Profile

```
# Quick profile with jq
cat movies.json | jq 'length'                          # Count
cat movies.json | jq '[.[].year] | unique | sort'       # Year range
cat movies.json | jq '[.[].rating | select(. == null)] | length'  # Null ratings
```

```
# Or with Python/pandas
df = pd.DataFrame(movies)
print(df.info())
print(df.describe())
print(df.isnull().sum())
```

Stage 3: Validate - Define Schema

```python
from jsonschema import validate, ValidationError

schema = {
    "type": "object",
    "properties": {
        "title": {"type": "string", "minLength": 1},
        "year": {"type": "integer", "minimum": 1880, "maximum": 2030},
        "rating": {"type": ["number", "null"]}
    },
    "required": ["title", "year"]
}
```

Stage 3: Validate - Run Validation

```python
valid_movies = []
invalid_movies = []

for movie in movies:
    try:
        validate(instance=movie, schema=schema)
        valid_movies.append(movie)
    except ValidationError as e:
        invalid_movies.append({"movie": movie, "error": str(e)})

print(f"Valid: {len(valid_movies)}, Invalid: {len(invalid_movies)}")
```

Stage 4: Clean and Transform

```python
def clean_movie(movie):
    """Transform raw movie data into clean format."""
    return {
        "title": movie["title"].strip(),
        "year": int(movie["year"]),
        "rating": float(movie["rating"]) if movie.get("rating") else None,
        "revenue": parse_revenue(movie.get("revenue")),
        "genres": parse_genres(movie.get("genre")),
    }
```

Stage 4: Helper Functions

```python
def parse_revenue(rev_str):
    """Convert '$292,576,195' to 292576195"""
    if not rev_str or rev_str == "N/A":
        return None
    return int(rev_str.replace("$", "").replace(",", ""))

# Apply cleaning to all valid movies
cleaned_movies = [clean_movie(m) for m in valid_movies]
```

Complete Pipeline Script (Part 1)

```bash
#!/bin/bash
# validate_movies.sh

INPUT=$1
OUTPUT_VALID="movies_valid.json"
OUTPUT_INVALID="movies_invalid.json"

echo "=== Stage 1: Ingest ==="
echo "Input file: $INPUT"
file "$INPUT"
cat "$INPUT" | jq 'length'
```

Complete Pipeline Script (Part 2)

```bash
echo -e "\n=== Stage 2: Profile ==="
cat "$INPUT" | jq '[.[].year | select(. == null)] | length'
cat "$INPUT" | jq '[.[].rating | select(. == null)] | length'

echo -e "\n=== Stage 3: Validate ==="
python validate.py "$INPUT" "$OUTPUT_VALID" "$OUTPUT_INVALID"

echo -e "\n=== Stage 4: Summary ==="
echo "Valid records: $(cat $OUTPUT_VALID | jq 'length')"
echo "Invalid records: $(cat $OUTPUT_INVALID | jq 'length')"
```

Pipeline Output

```
═══ Stage 1: Ingest ═══
Input file: movies_raw.json
movies_raw.json: JSON data, UTF-8 Unicode text
1000

═══ Stage 2: Profile ═══
Null years: 13
Null ratings: 108

═══ Stage 3: Validate ═══
Processing 1000 movies...
Valid: 879, Invalid: 121

═══ Stage 4: Summary ═══
Valid records: 879
Invalid records: 121

Validation complete. Check movies_invalid.json for details.
```

Back to Netflix: Cleaned Data

```
# Before cleaning
{"Title": "Inception", "Year": "2010", "imdbRating": "8.8",
 "BoxOffice": "$292,576,195", "Genre": "Action, Adventure, Sci-Fi"}

# After pipeline
{"title": "Inception", "year": 2010, "rating": 8.8,
 "revenue": 292576195, "genres": ["Action", "Adventure", "Sci-Fi"]}
```

Now we can train our model!

```
df = pd.DataFrame(cleaned_movies)
X = df[['year', 'rating']]  # Numeric columns
y = df['revenue']
model.fit(X, y)  # Works!
```

Part 12: Looking Ahead

Lab preview and next week

This Week's Lab

Hands-on Practice:

1. **Unix inspection** - `head`, `tail`, `wc`, `file`, `sort`, `uniq`
2. **jq exercises** - JSON querying and transformation
3. **CSVkit** - Profile and query CSV files
4. **Pydantic deep dive** - Nested models, custom validators
5. **Build a pipeline** - End-to-end validation of messy data

Goal: Take raw messy data and produce clean validated dataset.

Lab Dataset

You'll receive:

- `movies_raw.json` - 1000 movies with various quality issues
- `schema.json` - Partial schema (you'll complete it)

Issues to find and fix:

- Missing values (null, "N/A", empty string)
- Wrong types (numbers as strings)
- Duplicates
- Inconsistent formats
- Outliers

Next Week Preview

Week 3: Data Labeling

- Why labeling is the bottleneck
- Labeling tools and platforms
- Quality control for labels
- Inter-annotator agreement
- Managing labeling projects

The data we cleaned now needs labels for ML!

Interview Questions

Common interview questions on data validation:

1. **"How would you handle missing values in a dataset?"**
   - Identify types of missingness (MCAR, MAR, MNAR)
   - Strategies: deletion, imputation, flagging
   - Context matters: dropping vs filling depends on data and use case
2. **"What's the difference between validation and cleaning?"**
   - Validation: checking if data meets rules (returns true/false)
   - Cleaning: transforming data to meet rules (modifies data)
   - Validation should come first to understand the problems

Key Takeaways

1. **Look before you process** - Never trust raw data
2. **Know your enemy** - Understand types of data problems
3. **Tools matter** - jq, CSVkit, Pydantic save hours
4. **Schema-first** - Define expectations before processing
5. **Validate at the boundary** - Catch problems early
6. **Fail fast** - Don't propagate bad data
7. **Use Pydantic** - Pythonic validation with type hints

Resources

Tools:

- jq: https://stedolan.github.io/jq/manual/
- CSVkit: https://csvkit.readthedocs.io/
- Pydantic: https://docs.pydantic.dev/
- JSON Schema: https://json-schema.org/

Practice:

- jq playground: https://jqplay.org/

Questions?

Thank You!

See you in the lab!