Data Science & ML Projects (Beginner)

A Complete 8-Chapter Data Science & ML Course

Projects built:

A web scraper, a reusable data-cleaning pipeline, a live weather dashboard
A pandas-powered finance analyzer, a full movie-ratings EDA, a multi-API aggregator
A first real classifier, and a capstone combining every stage on your own dataset

Format: A4 · Dark-theme code examples, one complete project per chapter

Course 6 of 6 in the Data Science & ML subject

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 8

Web Scraper: Building a Local Library of Linux Cheatsheets

Data Science & ML Projects (Beginner)

Chapter 1 · Web Scraper: Building a Local Library of Linux Cheatsheets

> *Real data science work starts long before any model gets trained — most of it is collecting and organizing data that doesn't yet exist in a clean, usable form. This chapter is deliberately the one project in this course with no modeling at all: a reusable tool that collects every document in one category from a listing site and builds a local library out of them.*

What We're Building

A scraper that: fetches a category listing page, follows its pagination across every page in that category (not just the first, unlike `py4-7`'s own single-page version), visits each item's own detail page, downloads the linked content, and saves everything into a local folder plus a JSON manifest describing what was collected.

It's built and tested here against books.toscrape.com — the sister sandbox to `py4-7`'s own `quotes.toscrape.com`, from the same project, explicitly built and maintained for scraping practice. The tool itself is generic by design: point the finished script at a real, permitted Linux-cheatsheet site (after checking that site's own `robots.txt` and Terms of Service, per this chapter's own warn-box) and it collects cheatsheets exactly the same way it collects practice-site book listings here.

Step 1: Fetching One Category's Listing Page

```
import requests
from bs4 import BeautifulSoup

CATEGORY_URL = "https://books.toscrape.com/catalogue/category/books/travel_2/index.html"

response = requests.get(CATEGORY_URL)
soup = BeautifulSoup(response.text, "html.parser")
items = soup.find_all("article", class_="product_pod")
print(len(items))   # one
 per item on this page
```

Same `requests.get()` + `BeautifulSoup(...).find_all()` pattern `py4-7` already introduced — one page fetched, one set of matching elements found.

Step 2: Following Pagination Across the Whole Category

```
import time

def get_all_items(start_url):
    items = []
    url = start_url
    while url:
        response = requests.get(url)
        soup = BeautifulSoup(response.text, "html.parser")
        items.extend(soup.find_all("article", class_="product_pod"))

        next_link = soup.find("li", class_="next")
        url = requests.compat.urljoin(url, next_link.find("a")["href"]) if next_link else None
        time.sleep(1)   # one polite pause per page, per this chapter's own warn-box

    return items
```

A `while` loop keeps following the page's own `"Next →"` link (`soup.find("li", class_="next")`) until there isn't one — `None` ends the loop. `requests.compat.urljoin()` turns the relative link on the page into a full URL. `time.sleep(1)` between requests is the same discipline `py4-7`'s own warn-box named, applied here for real since this chapter actually fetches multiple pages.

Step 3: Visiting Each Item's Own Detail Page

```
def get_detail(item, base_url):
    link = item.find("h3").find("a")["href"]
    detail_url = requests.compat.urljoin(base_url, link)

    response = requests.get(detail_url)
    time.sleep(1)
    soup = BeautifulSoup(response.text, "html.parser")

    return {
        "title": soup.find("h1").get_text(),
        "price": soup.find("p", class_="price_color").get_text(),
        "url": detail_url,
    }
```

Each item on the listing page only has a title and a link — the fuller record lives on its own detail page. `get_detail()` follows that link and pulls out the specific fields worth keeping, exactly the "visit the linked page for the real content" step a real cheatsheet-collector would need.

Step 4: Saving the Manifest

```
import json
from pathlib import Path

def save_manifest(records, path="library_manifest.json"):
    Path(path).write_text(json.dumps(records, indent=2))
    print(f"Saved {len(records)} records to {path}.")
```

The same `pathlib` + `json` save pattern `py4-7` used — a manifest is just data, and it persists the same way any other collected data does.

The Complete Scraper

```
import json, time, requests
from bs4 import BeautifulSoup
from pathlib import Path

START_URL = "https://books.toscrape.com/catalogue/category/books/travel_2/index.html"

def get_all_items(start_url):
    items, url = [], start_url
    while url:
        soup = BeautifulSoup(requests.get(url).text, "html.parser")
        items.extend(soup.find_all("article", class_="product_pod"))
        next_link = soup.find("li", class_="next")
        url = requests.compat.urljoin(url, next_link.find("a")["href"]) if next_link else None
        time.sleep(1)
    return items

def get_detail(item, base_url):
    link = item.find("h3").find("a")["href"]
    detail_url = requests.compat.urljoin(base_url, link)
    soup = BeautifulSoup(requests.get(detail_url).text, "html.parser")
    time.sleep(1)
    return {
        "title": soup.find("h1").get_text(),
        "price": soup.find("p", class_="price_color").get_text(),
        "url": detail_url,
```

```
    }

items = get_all_items(START_URL)
records = [get_detail(item, START_URL) for item in items]
Path("library_manifest.json").write_text(json.dumps(records, indent=2))
print(f"Saved {len(records)} records.")
```

SCRAPING ETHICS, REVISITED FROM PY4-7

Every point `py4-7`'s own warn-box made still applies, and matters more here since this chapter fetches many pages, not one: check the target site's own `/robots.txt` and Terms of Service before scraping it, keep a real delay between requests (`time.sleep()`, used twice above — once per listing page, once per detail page), and never point this tool at a site that hasn't given permission. `books.toscrape.com` is used here specifically because it's built *for* this practice — that permission does not transfer automatically to a real Linux-cheatsheet site. Find one that explicitly allows scraping, or one offering its own downloadable archive/API, before reusing this exact script for the real goal.

Pagination loops

A `while` loop following a "Next" link until there isn't one.

requests.compat.urljoin()

Turns a relative link on a page into a full, fetchable URL.

Two-level scraping

Listing pages for links, detail pages for the real content.

Manifest files

A JSON record of exactly what a scraper collected and from where.

Extend This Project

Try these on your own:

- Point this exact script at a real, permitted Linux-cheatsheet source you've checked `robots.txt`/ToS for — this is the actual real-world goal the practice run above was rehearsing.
- Wrap each `requests.get()` call in `try`/`except` to skip a failed page instead of crashing the whole run.
- Save each item's own detail-page HTML (or linked file) to disk as a separate file, named from its own title, alongside the manifest — the actual "local library," not just metadata about one.
- Add a command-line argument (`argparse`) for the category URL, so the same script works against any category without editing the code.

What's Next

Chapter 2: **Data Cleaning Pipeline: Wrangling a Messy Real-World CSV** — turning `ds1-4` 's own cleaning toolkit into a real, reusable pipeline, run against a genuinely messy dataset instead of a small illustrative one.

CHAPTER 1 QUICK REFERENCE

- **Deliberately model-free** — data collection alone is real, substantial data science work, not a preliminary step to rush through
- Extends py4-7's own single-page scraper into a genuine multi-page, two-level (listing + detail) collection tool
- **Pagination:** follow "Next" links in a loop until none remain, with a polite delay per request
- Built and tested on books.toscrape.com's own safe practice sandbox; the real target (a permitted cheatsheet site) is the reader's own next step
- **Next chapter:** Data Cleaning Pipeline: Wrangling a Messy Real-World CSV

CHAPTER 2 OF 8

Data Cleaning Pipeline: Wrangling a Messy Real-World CSV

Data Science & ML Projects (Beginner)

Chapter 2 · Data Cleaning Pipeline: Wrangling a Messy Real-World CSV

> *`ds1-4` worked through cleaning as a one-time notebook exercise on one small, illustrative dataset. This chapter turns that same toolkit — missing values, duplicates, type conversion, string cleaning, outliers — into a real, reusable pipeline: a set of functions that can be pointed at any messy CSV and will both clean it and explain, in plain language, exactly what it did.*

What We're Building

A `clean_pipeline(df)` function that takes a raw, messy customer-orders CSV, returns a cleaned DataFrame, and generates a human-readable cleaning report — a running log of every fix it made and why.

```
# customer_orders.csv — real-world messy, on purpose
order_id,customer_name,email,order_date,amount,country
1001," john smith ",john@example.com,2026-01-04,"$45.00",USA
1002,JANE DOE,jane@example.com,01/05/2026,"$120",uk
1003,Bob Lee,bob@example.com,2026-01-06,,USA
1001," john smith ",john@example.com,2026-01-04,"$45.00",USA
1004,Amy Chen,amy@example.com,2026-01-07,"$999999.99",
1005,dan miller,,2026-01-08,"eror",USA
```

Step 1: Load & Inspect First

```
import pandas as pd

df = pd.read_csv("customer_orders.csv")
print(df.info())
print(df.isna().sum())
print(df.duplicated().sum())
```

Same diagnostic reflex `ds1-3` / `ds1-4` already established — `.info()` for dtypes, `.isna().sum()` for missing values per column, `.duplicated().sum()` for exact duplicate rows. Never clean before looking; the

shape of the mess determines which fixes actually apply.

Step 2: Duplicates

```
def remove_duplicates(df, report):
    before = len(df)
    df = df.drop_duplicates()
    removed = before - len(df)
    report.append(f"Removed {removed} exact duplicate row(s).")
    return df
```

Every cleaning function here follows the same shape: do the fix, measure what changed, append a plain-English line to a shared `report` list. Row `1001` above is an exact duplicate — same order, same everything — the safest kind of row to simply drop.

Step 3: Fix Types & Formats

```
def fix_amount(df, report):
    before_missing = df["amount"].isna().sum()
    df["amount"] = (
        df["amount"].astype(str)
        .str.replace("$", "", regex=False)
    )
    df["amount"] = pd.to_numeric(df["amount"], errors="coerce")   # "eror" -> NaN
    coerced = df["amount"].isna().sum() - before_missing
    report.append(f"Converted amount to numeric; {coerced} unparseable value(s) became missing.")
    return df

def fix_dates(df, report):
    df["order_date"] = pd.to_datetime(df["order_date"], errors="coerce")
    report.append("Standardized order_date to a single datetime format.")
    return df
```

`errors="coerce"` is the load-bearing choice in both functions — instead of crashing on `"eror"` or an inconsistent date format, pandas turns anything it can't parse into `NaN`, which Step 5 then handles deliberately rather than letting a bad value silently corrupt a calculation.

Step 4: Standardize Text

```
def standardize_names(df, report):
    df["customer_name"] = df["customer_name"].str.strip().str.title()
    df["country"] = df["country"].str.strip().str.upper()
    report.append("Trimmed whitespace and standardized casing on customer_name and country.")
    return df
```

`" john smith "` and `"JANE DOE"` are two customers written two very different ways — `.str.strip()` removes stray whitespace, `.str.title()` gives every name the same casing convention, so two records for the same person don't silently look like two different customers to anything downstream.

Step 5: Missing Values — A Real Decision, Per Column

```
def handle_missing(df, report):
    df = df.dropna(subset=["customer_name"])   # can't analyze an order with no customer
    report.append("Dropped rows with a missing customer_name (unrecoverable identifier).")

    median_amount = df["amount"].median()
    n_filled = df["amount"].isna().sum()
    df["amount"] = df["amount"].fillna(median_amount)
    report.append(f"Filled {n_filled} missing amount value(s) with the median
({median_amount:.2f}).")

    n_unknown = df["country"].isna().sum()
    df["country"] = df["country"].fillna("UNKNOWN")
    report.append(f"Filled {n_unknown} missing country value(s) with 'UNKNOWN'.")
    return df
```

EVERY FILL DECISION HERE IS A JUDGMENT CALL, NOT A CERTAINTY

Per `ds1-4`, there's no universally correct way to handle a missing value — dropping loses a whole row's worth of other good data; filling invents a value that was never actually observed. This pipeline drops rows missing a customer name (nothing useful can be salvaged), but fills missing amounts with the median (a reasonable estimate) and missing countries with an explicit `"UNKNOWN"` label rather than a silent guess. A different dataset, or a different downstream use, could reasonably justify different choices — which is exactly why the report logs each decision instead of making it invisibly.

Step 6: Flagging (Not Silently Removing) Outliers

```
def flag_outliers(df, report):
    q1, q3 = df["amount"].quantile([0.25, 0.75])
    iqr = q3 - q1
```

```
    upper_bound = q3 + 1.5 * iqr   # the same IQR rule ds1-6 covers in full
    flagged = df[df["amount"] > upper_bound]
    report.append(f"Flagged {len(flagged)} row(s) as statistical outliers (amount > ${upper_bound:.2f}) for manual review — not removed.")
    return df, flagged
```

`$999999.99` is almost certainly a data-entry error, not a genuine order — but this pipeline flags it for a human to check rather than deleting it automatically. `ds1-6` covers the IQR rule's own full statistical reasoning; here it's applied as a practical, reusable check.

The Complete Pipeline

```
def clean_pipeline(input_path):
    df = pd.read_csv(input_path)
    report = [f"Started with {len(df)} rows."]

    df = remove_duplicates(df, report)
    df = fix_amount(df, report)
    df = fix_dates(df, report)
    df = standardize_names(df, report)
    df = handle_missing(df, report)
    df, flagged = flag_outliers(df, report)

    report.append(f"Finished with {len(df)} rows.")
    return df, flagged, report

cleaned, flagged, report = clean_pipeline("customer_orders.csv")
cleaned.to_csv("customer_orders_cleaned.csv", index=False)
print("\n".join(report))
```

NEVER OVERWRITE THE RAW FILE

The pipeline reads from `customer_orders.csv` and writes to a *new* file, `customer_orders_cleaned.csv`. Every cleaning decision here — the median fill, the "UNKNOWN" label, the outlier threshold — is a choice that could turn out to be wrong in hindsight. Keeping the original file untouched means any of these decisions can be revisited later without needing to re-collect the data.

errors="coerce"

Turns unparseable values into NaN instead of crashing the whole conversion.

Per-column missing-value strategy

Drop, fill, or label "UNKNOWN" — a real decision made once, per column, and logged.

Flag, don't silently delete

Self-documenting pipelines

Outliers get marked for review, not removed automatically.

Every cleaning function appends to a shared report — nothing happens invisibly.

Extend This Project

Try these on your own:

- Turn the printed report into a saved `cleaning_report.txt` file alongside the cleaned CSV, so the log survives the run that produced it.
- Add a function that standardizes email addresses (lowercase, strip whitespace) the same way `standardize_names` handles names.
- Make `clean_pipeline()` accept a config dict specifying which columns to drop-on-missing versus fill-on-missing, instead of hardcoding the choice per column.
- Run the pipeline against the flagged-outlier rows only, and decide by hand whether each one should be corrected, dropped, or kept as a genuine (if unusual) order.

What's Next

Chapter 3: **Weather Data Dashboard** — the first project pulling live data from an API instead of a static file, applying ds1-3's own JSON-handling material to real-time responses.

CHAPTER 2 QUICK REFERENCE

- Turns ds1-4's own one-time cleaning walkthrough into a real, reusable, function-based pipeline
- **Six steps:** duplicates → type/format fixes → text standardization → missing values → outlier flagging → a generated report
- `errors="coerce"` converts bad values to NaN instead of crashing — handled deliberately downstream, not silently
- Missing-value strategy is a real per-column judgment call, always logged, never invisible
- Outliers are flagged for review, not auto-deleted; the IQR rule itself is covered in full in ds1-6
- Always write cleaned output to a new file — never overwrite the raw source

CHAPTER 3 OF 8

Weather Data Dashboard: Pulling, Storing & Visualizing Live API Data

Data Science & ML Projects (Beginner)

Chapter 3 · Weather Data Dashboard: Pulling, Storing & Visualizing Live API Data

> *Every project so far has worked with data that already existed somewhere — a scraped page, a saved CSV. This chapter's data doesn't exist until the moment it's requested: a live weather API, parsed with `ds1-3`'s own JSON-handling skills applied to a live response instead of a file on disk, then charted with `ds1-7`'s and `ds1-8`'s own tools.*

What We're Building

A small tool that fetches an hourly weather forecast from Open-Meteo — a genuinely free, public weather API that needs no signup or API key, which is exactly why it's used here — turns the response into a DataFrame, appends it to a growing local dataset, and builds two small charts from it: a temperature-over-time line chart and a precipitation-by-hour bar chart.

Step 1: Making the API Request

```
import requests

params = {
    "latitude": 51.51,     # London, as an example — swap for any location
    "longitude": -0.13,
    "hourly": "temperature_2m,precipitation",
    "forecast_days": 2,
}
response = requests.get("https://api.open-meteo.com/v1/forecast", params=params)
print(response.status_code)   # 200 — success
```

Passing a `params` dict to `requests.get()` builds the full query-string URL automatically — no manual string concatenation needed. No API key here, unlike most weather services, which is exactly why this API was chosen for a beginner project: nothing to configure, sign up for, or accidentally commit to a public repository.

Step 2: Parsing JSON From a Live Response

```
data = response.json()
print(list(data["hourly"].keys()))
# ['time', 'temperature_2m', 'precipitation']
```

THE SAME JSON, ARRIVING A DIFFERENT WAY

`ds1-3` read JSON with `json.load(open(path))` — a file that already existed on disk. Here, `response.json()` does the identical parsing job, but on a response body that only came into existence the moment this script made the request. The structure once parsed is identical (nested dicts and lists); only its source differs.

Step 3: A Genuinely Different JSON Shape

```
import pandas as pd

hourly = data["hourly"]
df = pd.DataFrame({
    "time": pd.to_datetime(hourly["time"]),
    "temperature": hourly["temperature_2m"],
    "precipitation": hourly["precipitation"],
})
print(df.head())
```

AN HONEST STRUCTURAL DIFFERENCE FROM EARLIER JSON

Open-Meteo returns *parallel lists* — one list of every timestamp, one list of every temperature, in matching order — rather than a list of individual row-like records the way most of this course's own earlier JSON examples looked. Building a DataFrame here means passing a dict of equal-length lists directly to `pd.DataFrame()`, not looping over a list of row dicts. Real APIs don't all agree on one shape; reading the actual response before assuming its structure is the real skill.

Step 4: Storing Results Locally Over Time

```
from pathlib import Path

def append_to_log(df, path="weather_log.csv"):
    if Path(path).exists():
        df.to_csv(path, mode="a", header=False, index=False)
    else:
```

```
        df.to_csv(path, index=False)
    print(f"Appended {len(df)} rows to {path}.")
```

Running this script once a day builds a genuinely growing local weather history out of nothing but repeated forecast pulls — `mode="a"` appends rather than overwriting, and skipping the header on every call after the first keeps the CSV valid.

Step 5: Building the Dashboard

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6))

ax1.plot(df["time"], df["temperature"], color="#15803D")
ax1.set_title("Temperature Forecast")
ax1.set_ylabel("°C")

ax2.bar(df["time"], df["precipitation"], color="#EAB308")
ax2.set_title("Precipitation Forecast")
ax2.set_ylabel("mm")

plt.tight_layout()
plt.savefig("weather_dashboard.png")
```

`ds1-7`'s own line-chart-for-a-trend and bar-chart-for-discrete-values choices, applied directly — temperature is a continuous value changing over time (a line), precipitation at each hour is a discrete quantity (a bar). Two subplots stacked in one figure make a small, genuine dashboard rather than two disconnected charts.

The Complete Script

```
import requests, pandas as pd, matplotlib.pyplot as plt
from pathlib import Path

params = {"latitude": 51.51, "longitude": -0.13,
          "hourly": "temperature_2m,precipitation", "forecast_days": 2}
data = requests.get("https://api.open-meteo.com/v1/forecast", params=params).json()

hourly = data["hourly"]
df = pd.DataFrame({
    "time": pd.to_datetime(hourly["time"]),
    "temperature": hourly["temperature_2m"],
```

```
    "precipitation": hourly["precipitation"],
})

log_path = "weather_log.csv"
df.to_csv(log_path, mode="a" if Path(log_path).exists() else "w",
          header=not Path(log_path).exists(), index=False)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6))
ax1.plot(df["time"], df["temperature"], color="#15803D")
ax1.set_title("Temperature Forecast")
ax2.bar(df["time"], df["precipitation"], color="#EAB308")
ax2.set_title("Precipitation Forecast")
plt.tight_layout()
plt.savefig("weather_dashboard.png")
print("Dashboard updated.")
```

BEING A GOOD CITIZEN OF A FREE PUBLIC API

Open-Meteo is free specifically for reasonable use — it isn't an invitation to poll it every second. Running this script on a schedule (once an hour, once a day) via a real scheduler is the appropriate pattern, not a tight loop. The same courtesy from Chapter 1's own scraping-ethics warn-box applies to APIs too: check a service's documented rate limits and terms before building anything that calls it repeatedly.

Query parameters as a dict

`requests.get(url, params={...})` builds the URL automatically.

response.json()

Parses a live response body the same way ds1-3 parsed a saved file.

Parallel-list JSON

Some APIs return matching lists instead of row-like records — read before assuming.

Appending, not overwriting

`mode="a"` builds a real local history out of repeated small pulls.

Extend This Project

Try these on your own:

- Add a second location and plot both on the same temperature chart, with a legend, to compare two cities at once.
- Wrap the request in `try` / `except` so a network failure logs a message instead of crashing the whole script.
- Once `weather_log.csv` has several days of real history, load it back in and plot a week-long trend instead of just the latest forecast.

- Use a Seaborn heatmap (`ds1-8`) to show temperature by hour-of-day across several logged days at once.

What's Next

Chapter 4: **Personal Finance Analyzer, Revisited With pandas** — taking `py4-8`'s own plain-Python expense tracker and rebuilding its analysis with the pandas/numpy toolkit this course has been using all along.

CHAPTER 3 QUICK REFERENCE

- The first project pulling from a live API rather than a static file — data that doesn't exist until requested
- **requests.get(url, params={...})** builds a query-string URL automatically; **.json()** parses the live response, same job as ds1-3's own json.load()
- Real APIs don't share one JSON shape — Open-Meteo's own parallel-list structure required a genuinely different DataFrame-building approach
- Appending (`mode="a"`) to a CSV over repeated runs builds a real local dataset out of small live pulls
- ds1-7's line-vs-bar chart choice applied directly: continuous trend (temperature) as a line, discrete quantity (precipitation) as a bar
- Free APIs still have limits — schedule pulls reasonably, don't poll in a tight loop

CHAPTER 4 OF 8

Personal Finance Analyzer, Revisited With pandas

Data Science & ML Projects (Beginner)

Chapter 4 · Personal Finance Analyzer, Revisited With pandas

> *`py4-8`'s own expense tracker built its category totals with a hand-rolled dict-accumulation loop and its overall total with a generator expression — genuinely good Python, and completely appropriate for that course's own scope. This chapter takes the exact same problem and rebuilds its analysis with the tools this course has been using all along, making concrete exactly what "the right tool for the job" bought a learner who has now taken both courses.*

What We're Building

The same expense data `py4-8` tracked — amount, category, description — now with one addition a real finance tool needs: a date per expense. Loaded into a DataFrame, the exact same two summaries `py4-8` built by hand become one-line pandas operations, and two genuinely new analyses (monthly trends, a real chart) become possible that weren't practical to hand-roll before.

```
# expenses.json — py4-8's own schema, plus one new field: date
[
  {"date": "2026-01-03", "amount": 12.50, "category": "Food", "description": "Lunch"},
  {"date": "2026-01-05", "amount": 45.00, "category": "Transport", "description": "Fuel"},
  {"date": "2026-02-01", "amount": 89.99, "category": "Food", "description": "Groceries"}
]
```

Step 1: Loading Into a DataFrame

```
import pandas as pd

df = pd.read_json("expenses.json")
df["date"] = pd.to_datetime(df["date"])
print(df.head())
```

One line replaces `py4-8`'s own `json.loads(FILE.read_text())` plus manually keeping the result as a list of dicts — `pd.read_json()` parses the file directly into a DataFrame.

Step 2: Totals — The Direct Payoff

PY4-8'S OWN PLAIN PYTHON	THIS CHAPTER'S PANDAS EQUIVALENT

`totals = {}`
`for e in expenses:`
`  totals[e["category"]] = (`
`  totals.get(e["category"], 0)`
`  + e["amount"]`
`  )`
	`totals = df.groupby("category")["amount"].sum()`

`sum(e["amount"] for e in expenses)`	`df["amount"].sum()`

`py4-8`'s own `totals.get(category, 0) + amount` pattern was itself a direct reuse of the site's own word-frequency-counter idiom — real, working code. `.groupby("category")["amount"].sum()` does the identical job in one call, built on the same vectorized, C-level operations `ds1-2` introduced as NumPy's own real advantage over a plain Python loop.

Step 3: What py4-8 Genuinely Couldn't Do Easily — Monthly Trends

```
monthly = df.set_index("date").resample("ME")["amount"].sum()
print(monthly)
# 2026-01-31    57.50
# 2026-02-28    89.99
```

Reproducing this in plain Python would mean manually parsing every date string, bucketing entries by year-and-month into a dict, and keeping that bucketing logic in sync everywhere it's needed — not impossible, but real, fiddly work `py4-8`'s own scope never asked for. `.resample("ME")` does the entire "group by calendar month" operation in one call, because the DataFrame's own datetime index already understands calendar structure.

Step 4: A Real Chart

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4.5))

totals.plot(kind="pie", ax=ax1, autopct="%1.0f%%", ylabel="",
            colors=["#15803D", "#EAB308", "#4ade80", "#facc15"])
ax1.set_title("Spending by Category")
```

```
monthly.plot(kind="line", marker="o", ax=ax2, color="#15803D")
ax2.set_title("Spending Over Time")
ax2.set_ylabel("$")

plt.tight_layout()
plt.savefig("finance_dashboard.png")
```

Both charts plot directly off the `groupby` / `resample` results from Steps 2 and 3 — a genuine payoff of keeping the data in a DataFrame throughout, rather than converting back to plain lists and dicts just to hand them to a chart.

The Complete Analysis Script

```
import pandas as pd, matplotlib.pyplot as plt

df = pd.read_json("expenses.json")
df["date"] = pd.to_datetime(df["date"])

totals = df.groupby("category")["amount"].sum()
monthly = df.set_index("date").resample("ME")["amount"].sum()

print("Totals by category:\n", totals)
print("\nMonthly spending:\n", monthly)
print(f"\nOverall total: ${df['amount'].sum():.2f}")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4.5))
totals.plot(kind="pie", ax=ax1, autopct="%1.0f%%", ylabel="")
ax1.set_title("Spending by Category")
monthly.plot(kind="line", marker="o", ax=ax2)
ax2.set_title("Spending Over Time")
plt.tight_layout()
plt.savefig("finance_dashboard.png")
```

HONEST NOTE: THIS ISN'T A BLANKET UPGRADE

For `py4-8`'s own scope — a few dozen expenses, entered one at a time through a CLI menu — plain Python was completely appropriate; pulling in pandas for that would have been real, unnecessary overhead. pandas earns its keep specifically at the scale and shape of analysis this chapter adds: month-over-month trends across a genuinely large history, and charts built directly from the data without a manual conversion step. The right tool depends on what the job actually is, not on which tool is newer or more powerful in the abstract.

pd.read_json()

.groupby().sum()

Loads JSON directly into a DataFrame — one line, no manual list-of-dicts handling.

Replaces a hand-rolled dict-accumulation loop with one vectorized call.

.resample("ME")

Groups by calendar month using a datetime index — no manual date-bucketing logic.

Charting straight from results

groupby/resample output plots directly — no conversion back to plain Python first.

Extend This Project

Try these on your own:

- Add a `budget` dict per category and compute how far over/under budget each category is for the current month.
- Use `.resample("W")` instead of `"ME"` to see weekly rather than monthly spending trends.
- Merge in the weather log from Chapter 3 by date and check, out of genuine curiosity, whether spending correlates with rainy days.
- Bring back `py4-8`'s own `add_expense()` CLI function to append new entries, then re-run this chapter's analysis on the growing file.

What's Next

Chapter 5: **Movie Ratings Explorer** — the course's own central EDA chapter, applying ds1-9's/ds1-10's own seven-step methodology to a fresh public dataset without a checklist to lean on.

CHAPTER 4 QUICK REFERENCE

- Rebuilds py4-8's own expense-tracker analysis with pandas — same problem, the tools this course has been using since Chapter 2
- `.groupby("category")["amount"].sum()` replaces a hand-rolled dict-accumulation loop directly
- `.resample("ME")` makes month-over-month trends practical in one line — genuinely hard to hand-roll well in plain Python
- Charts plot directly from groupby/resample results — no manual conversion step needed
- **Honest note:** pandas earns its keep at this chapter's own scale and shape of analysis, not universally — py4-8's own plain-Python scope didn't need it

CHAPTER 5 OF 8

Movie Ratings Explorer: A Full EDA on a Public Dataset

Data Science & ML Projects (Beginner)

Chapter 5 · Movie Ratings Explorer: A Full EDA on a Public Dataset

DELIBERATELY NO NUMBERED CHECKLIST THIS TIME

`ds1-9` walked through EDA as seven explicit, numbered steps — shape, summary stats, univariate, bivariate, multivariate, anomalies, hypotheses. Every one of those concerns is still here in this chapter, but presented the way an analyst actually uses them once the checklist has done its job: as instincts guiding a real investigation, not a list to tick off in order.

What We're Building

An exploration of a MovieLens-style ratings dataset — a real, well-known public research dataset of user ratings for movies, freely available and widely used for exactly this kind of practice. Two files: `movies.csv` (movie ID, title, genre, release year) and `ratings.csv` (user ID, movie ID, rating, timestamp). The question driving the whole exploration: what actually makes a movie "highly rated," and does the obvious answer hold up once the data gets a real look?

Loading and Merging

```
import pandas as pd

movies = pd.read_csv("movies.csv")
ratings = pd.read_csv("ratings.csv")

df = ratings.merge(movies, on="movie_id")
print(df.shape)
print(df.head())
print(df["rating"].describe())
```

The merge itself is `ds1-5`'s own join material put to real use — a ratings file alone has no genre or title, and a movies file alone has no ratings; the interesting questions only exist once both are joined. `.shape` and `.describe()` here aren't a separate "Step 1" — they're just the first thing worth checking before trusting anything built on top of this data.

What Does a Rating Actually Look Like?

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.histplot(df["rating"], bins=10, color="#15803D")
plt.title("Distribution of Individual Ratings")
plt.savefig("rating_distribution.png")
```

Ratings cluster heavily around 3-4 out of 5, with genuinely low ratings comparatively rare — a real, common pattern in rating data worth noticing before trusting any average built from it: the scale isn't used symmetrically, so "average rating" already means something skewed toward the positive end before any other analysis happens.

Which Genres Rate Highest?

```
genre_avg = df.groupby("genre")["rating"].mean().sort_values(ascending=False)
print(genre_avg)

genre_avg.plot(kind="barh", color="#EAB308")
plt.title("Average Rating by Genre")
plt.savefig("genre_ratings.png")
```

A one-line `groupby`, but already worth a second look before drawing conclusions — some genres will naturally have far fewer total ratings than others, which is exactly the kind of thing that turns into a genuine trap two sections from now.

Popular vs. Highly Rated — Not the Same Question

```
title_stats = df.groupby("title").agg(
    avg_rating=("rating", "mean"),
    num_ratings=("rating", "count"),
)

print("Top 10 by average rating:")
print(title_stats.sort_values("avg_rating", ascending=False).head(10))

print("\nTop 10 by number of ratings:")
print(title_stats.sort_values("num_ratings", ascending=False).head(10))
```

A REAL, COMMON FINDING

Sorted purely by average rating, the top of the list is dominated by obscure titles with only one or two ratings — a single perfect score from one viewer outranks a genuinely acclaimed film with thousands of ratings averaging 4.3. Sorted by number of ratings instead, a completely different, far more recognizable list appears. "Highest rated" and "most popular" are answering two different questions, and naively trusting the first without checking `num_ratings` produces a genuinely misleading result.

A Statistical Trap: Small-Sample Extremes

Why does a one-rating movie so easily top the list? With only one or two data points, the average has nothing to pull it toward a typical value — a single 5 or a single 1 *is* the entire average. As the number of ratings grows, the law of large numbers (`ds1-6`) pulls the average toward the movie's own genuine, underlying quality, since the extremes of individual opinion start to cancel out. This is the same small-sample-size caution `ds1-6` raised in the abstract, now caught red-handed in a real dataset.

```
reliable = title_stats[title_stats["num_ratings"] >= 20]
print(reliable.sort_values("avg_rating", ascending=False).head(10))
```

A REAL-WORLD PARALLEL

Filtering to a minimum rating count before ranking is a simplified version of exactly what real rating platforms (IMDb among them) do — a raw average alone is known to be unreliable at low sample sizes, so production systems use some form of minimum-count threshold or weighted formula rather than trusting a naive `.mean()` directly.

Forming a Real Hypothesis: Do Older Movies Rate Higher?

```
df["decade"] = (df["year"] // 10) * 10
decade_avg = df.groupby("decade")["rating"].mean()

decade_avg.plot(kind="line", marker="o", color="#15803D")
plt.title("Average Rating by Decade")
plt.savefig("decade_trend.png")
```

A PATTERN ISN'T A CONCLUSION

If older decades show a slightly higher average, resist the urge to conclude "movies used to be better." `ds1-6` 's own correlation-vs-causation warning applies directly here: only movies good enough to still be watched and rated today tend to survive into a modern ratings dataset at all — a real selection-bias effect, not evidence about film quality across all eras equally. The honest conclusion is a genuine, testable hypothesis worth investigating further, not a settled finding.

Merging before exploring

The interesting questions only exist once both files are joined (ds1-5).

groupby + agg with named aggregations

Computing average and count together, side by side, in one call.

Small-sample extremes

Low-count groups produce unreliable, easily-extreme averages — filter before ranking.

Selection bias in survivorship

What's still in the dataset today isn't a random sample of everything that ever existed.

Extend This Project

Try these on your own:

- Try a few different minimum-`num_ratings` thresholds (5, 20, 50) and see how much the top-10 list changes at each one.
- Compute each genre's own average *and* total rating count together, the same way this chapter did for titles, before trusting the genre bar chart's own ranking.
- Look up whether a Bayesian-average or weighted-rating formula (search "IMDb weighted rating formula" for a real, documented example) produces a meaningfully different ranking than the simple threshold used here.
- Plot ratings-per-year instead of average-rating-per-year — does the dataset simply contain more *recent* ratings, which could itself explain part of the decade pattern?

What's Next

Chapter 6: **Multi-Source Data Aggregator** — combining two independent public APIs into one clean, merged dataset, extending Chapter 3's own single-API pattern to genuine multi-source integration.

CHAPTER 5 QUICK REFERENCE

- Applies ds1-9's/ds1-10's own seven EDA concerns as a flowing investigation, not a numbered checklist — the shape/stats/univariate/bivariate/anomaly/hypothesis concerns are all still present
- Merging (ds1-5) two files first, since the interesting questions only exist once both are joined
- "Highest rated" and "most popular" are genuinely different questions — a naive average alone can be misleading
- Small-sample extremes are a real statistical trap (ds1-6), directly caught in this chapter's own top-10 list

- A visible pattern (older movies rating higher) is a hypothesis worth investigating, not a conclusion — selection/survivorship bias is a real, honest alternative explanation

CHAPTER 6 OF 8

Multi-Source Data Aggregator: Combining Two Public APIs Into One Clean Dataset

Data Science & ML Projects (Beginner)

Chapter 6 · Multi-Source Data Aggregator: Combining Two Public APIs Into One Clean Dataset

> *Chapter 3 fetched from one API. This chapter fetches from two — independent services, built by different teams, sharing no obvious common key — and combines them into one clean dataset using `ds1-5`'s own merge material, applied to a real join key that has to be derived rather than one that's simply handed to you.*

What We're Building

A dataset combining country information from the REST Countries API (population, capital, currency — free, no signup) with live currency exchange rates from an open exchange-rate API (also free, no signup) — producing one table showing each country's population alongside its own currency's current value against the US dollar. Neither API alone can answer "which populous countries currently have the weakest currency" — only the combination can.

Step 1: Fetching From the First API

```
import requests
import pandas as pd

response = requests.get(
    "https://restcountries.com/v3.1/region/europe",
    params={"fields": "name,capital,population,currencies"},
)
countries_data = response.json()
print(len(countries_data), "countries returned")
```

Same `requests.get(url, params={...})` pattern Chapter 3 introduced — just a different service, restricted to one region to keep this example a manageable size.

Step 2: Extracting a Currency Code From Nested JSON

```
records = []
for c in countries_data:
    currencies = c.get("currencies", {})
    currency_code = list(currencies.keys())[0] if currencies else None
    records.append({
        "country": c["name"]["common"],
        "capital": c.get("capital", [None])[0],
        "population": c["population"],
        "currency_code": currency_code,
    })

countries_df = pd.DataFrame(records)
print(countries_df.head())
```

Each country's currency arrives as a nested dict keyed by its own three-letter code (e.g. `{"EUR": {...}}`) — `currency_code` here isn't a field this API hands over directly, it's derived by pulling the first key out of that nested structure. This derived value is exactly what makes the merge in Step 4 possible at all.

Step 3: Fetching From the Second, Unrelated API

```
rates_response = requests.get("https://open.er-api.com/v6/latest/USD")
rates_data = rates_response.json()["rates"]

rates_df = pd.DataFrame(list(rates_data.items()), columns=["currency_code", "usd_rate"])
print(rates_df.head())
```

This API has never heard of REST Countries and doesn't organize its data by country at all — it's a flat dict of currency codes to exchange rates. `rates_data.items()` turns that dict into a list of (code, rate) pairs, which becomes a two-column DataFrame — the same currency-code column `countries_df` already has, arrived at completely independently.

Step 4: The Merge — ds1-5's Own Material, on a Derived Key

```
combined = countries_df.merge(rates_df, on="currency_code", how="left")
print(combined.sort_values("population", ascending=False).head(10))
```

WHY HOW="LEFT", AND WHAT IT HONESTLY REVEALS

`how="left"` keeps every country from `countries_df` even if its currency code has no match in `rates_df` — some currency codes are obscure enough that a live exchange-rate feed simply doesn't track them. Those rows end up with `NaN` in `usd_rate`, exactly the missing-value situation `ds1-4`'s own cleaning material covers — a real, honest consequence of combining two independently-maintained data sources rather than a mistake in this chapter's own code.

A Quick Combined Insight

```
top_by_population = combined.sort_values("population", ascending=False).head(10)
print(top_by_population[["country", "population", "currency_code", "usd_rate"]])
```

A table like this — population from one API, live currency strength from a completely different one — genuinely doesn't exist as a single downloadable file anywhere. It only exists because this chapter built it, on the spot, from two independent sources.

The Complete Aggregator

```
import requests, pandas as pd

countries_data = requests.get(
    "https://restcountries.com/v3.1/region/europe",
    params={"fields": "name,capital,population,currencies"},
).json()

records = []
for c in countries_data:
    currencies = c.get("currencies", {})
    records.append({
        "country": c["name"]["common"],
        "capital": c.get("capital", [None])[0],
        "population": c["population"],
        "currency_code": list(currencies.keys())[0] if currencies else None,
    })
countries_df = pd.DataFrame(records)

rates_data = requests.get("https://open.er-api.com/v6/latest/USD").json()["rates"]
rates_df = pd.DataFrame(list(rates_data.items()), columns=["currency_code", "usd_rate"])

combined = countries_df.merge(rates_df, on="currency_code", how="left")
combined.to_csv("country_currency_combined.csv", index=False)
print(f"Combined {len(combined)} countries across two independent APIs.")
```

NOW CONSIDERATE OF TWO SERVICES, NOT ONE

Chapter 3's own API-courtesy point applies doubled here — this script makes exactly one call to each API per run, not one call per country, which is exactly why the second API was chosen because it returns every currency's rate in a single response rather than requiring a separate call per country. Combining sources doesn't have to mean multiplying request volume; sometimes the right design choice avoids the problem entirely.

Independently-shaped sources

Two APIs, two different JSON structures, no shared convention between them.

Deriving a join key

currency_code isn't handed over directly — it's pulled from nested JSON first.

how="left" and its honest cost

Keeps every row from the primary source; unmatched keys become real, meaningful NaNs.

Request-count-aware design

One call per API, not one call per row — considerate by design, not just by delay.

Extend This Project

Try these on your own:

- Change the region to `"asia"`, `"africa"`, or `"americas"` and compare how many currency codes fail to find a match in each region.
- Use ds1-4's own cleaning toolkit to decide, explicitly, what to do with the countries that ended up with a missing `usd_rate` — drop them, or leave them flagged.
- Add a third source: a real, free country-flag or ISO-code API, merged in on a different derived key.
- Build a scatter plot of population vs. `usd_rate` and look, cautiously (per Chapter 5's own selection-bias warning), for any pattern worth investigating further rather than concluding one exists.

What's Next

Chapter 7: **A First Real Classifier** — this course's own single, deliberately light touch of ml1, a small, approachable classification project on a well-known dataset.

CHAPTER 6 QUICK REFERENCE

- Extends Chapter 3's own single-API pattern to two independent APIs, combined via ds1-5's own merge material
- A join key can be derived (pulled out of nested JSON) rather than handed over directly by either source

- `how="left"` preserves the primary source's own rows; unmatched keys become real, honest missing values (ds1-4)
- Choosing an API that returns all needed data in one call avoids one-request-per-row volume entirely
- The resulting combined table doesn't exist anywhere as a single downloadable file — it's only produced by the combination itself

CHAPTER 7 OF 8

A First Real Classifier: A Light Touch of scikit-learn

Data Science & ML Projects (Beginner)

Chapter 7 · A First Real Classifier: A Light Touch of scikit-learn

> *Six chapters, zero models — deliberately, per this course's own opening throughline. This one chapter is the exception: a small, honest taste of what `ml1` covers in full depth, not a substitute for that course. Every shortcut this chapter takes is named explicitly, not smoothed over.*

What We're Building

A classifier for the Iris dataset — the single most famous "hello world" of classification: predict which of three flower species a sample belongs to, from four simple measurements (sepal length, sepal width, petal length, petal width). It ships built directly into scikit-learn, needs zero cleaning, and is small enough to see the whole pipeline at once.

Step 1: Loading the Dataset

```
from sklearn.datasets import load_iris
import pandas as pd

iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df["species"] = [iris.target_names[i] for i in iris.target]
print(df.head())
```

`iris.data` is a NumPy array of the four measurements; `iris.target` is the numeric species label for each row (0, 1, or 2). Building a DataFrame and mapping the numbers back to real species names makes the data readable the same way every prior chapter's own data has been.

Step 2: A Quick Look — Why This Dataset Works So Well

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
sns.scatterplot(
    data=df, x="petal length (cm)", y="petal width (cm)",
    hue="species", palette=["#15803D", "#EAB308", "#4ade80"],
)
plt.title("Petal Measurements by Species")
plt.savefig("iris_scatter.png")
```

Plotting just two of the four features already shows the three species separating into clearly distinct clusters — a genuine, visible reason this particular dataset became the classic teaching example: the classes are almost perfectly separable, letting even a simple model succeed clearly enough to see what's actually happening.

Step 3: Train/Test Split

```
from sklearn.model_selection import train_test_split

X = df[iris.feature_names]
y = df["species"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print(f"Training on {len(X_train)} rows, testing on {len(X_test)} rows.")
```

The full reasoning behind why a model needs to be evaluated on data it never saw during training is `ml1-2`'s own material in full — used correctly here, not re-derived. `random_state=42` just makes the split reproducible from one run to the next.

Step 4: Fitting a Simple Classifier

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
```

A DELIBERATELY DIFFERENT ALGORITHM THAN ML1 COVERS

K-Nearest Neighbors classifies a new sample by looking at its `n_neighbors` closest points in the training data and taking a majority vote of their species — genuinely intuitive, no gradient descent or decision-tree splitting logic to explain. It's used here specifically because it's *not* logistic regression (`ml1-5`) or decision trees/random forests (`ml1-7`) — those get their own full, proper treatment in that course; this chapter isn't trying to preempt either one.

Step 5: Evaluating — The Single Simplest Metric

```
accuracy = model.score(X_test, y_test)
print(f"Accuracy: {accuracy:.2%}")
```

ACCURACY ALONE IS THE SIMPLIFIED VERSION

`.score()` here reports plain accuracy — the fraction of test predictions that were correct. It's genuinely the simplest possible metric, and genuinely not the full picture: `ml1-6` covers precision, recall, F1, and the confusion matrix in depth, including exactly why accuracy alone can be misleading on imbalanced data. Iris happens to have equal numbers of each species, which is part of why plain accuracy is defensible here specifically — that won't be true of every dataset.

Step 6: Seeing the Predictions

```
predictions = model.predict(X_test)
results = X_test.copy()
results["actual"] = y_test.values
results["predicted"] = predictions
results["correct"] = results["actual"] == results["predicted"]

print(results[results["correct"] == False])   # the rows the model got wrong, if any
```

Looking directly at whichever rows the model got wrong (if any) is worth more than the single accuracy number alone — on this dataset, mistakes almost always happen between two of the three species that sit closest together in the Step 2 scatter plot, not randomly across all three, a small, concrete confirmation that the model is making genuinely sensible errors rather than arbitrary ones.

The Complete Classifier

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
import pandas as pd

iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df["species"] = [iris.target_names[i] for i in iris.target]

X = df[iris.feature_names]
```

```
y = df["species"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

accuracy = model.score(X_test, y_test)
print(f"Accuracy: {accuracy:.2%}")
```

Built-in datasets

load_iris() ships with scikit-learn — no download, no cleaning needed.

train_test_split()

Used correctly here; ml1-2 covers the full reasoning behind it.

K-Nearest Neighbors

Classify by majority vote among the closest training points — no gradient descent.

Accuracy's own honest limits

The simplest metric, not the full picture — ml1-6 covers precision/recall/F1.

Extend This Project

Try these on your own:

- Change `n_neighbors` to 1 and to 15 — how does accuracy change, and why might too few or too many neighbors both cause problems?
- Once you've taken ml1, come back and fit a `LogisticRegression` or `DecisionTreeClassifier` on this exact same data and compare its accuracy to KNN's own.
- Use only two features (say, petal length and width) instead of all four, and see how much accuracy actually drops.
- Try the same pipeline on scikit-learn's other built-in dataset, `load_wine()`, with no other changes — does it still work this cleanly?

What's Next

Chapter 8: **Capstone — Your Own Dataset, Start to Finish** — no fixed dataset this time: your own choice, taken through the complete collect-clean-explore-visualize-model pipeline this course has built one piece at a time.

CHAPTER 7 QUICK REFERENCE

- This course's own single, deliberately light touch of ml1 — a taste, not a substitute for that course's own full depth
- Iris: built into scikit-learn, needs no cleaning, chosen specifically to keep the focus on the modeling pipeline itself
- K-Nearest Neighbors used deliberately instead of logistic regression (ml1-5) or decision trees (ml1-7), so as not to preempt either
- `train_test_split()` used correctly, not re-derived — ml1-2 covers the full reasoning
- Plain accuracy reported honestly as the simplest metric, not the full evaluation picture ml1-6 covers

CHAPTER 8 OF 8

Capstone: Your Own Dataset, Start to Finish

Data Science & ML Projects (Beginner)

Chapter 8 · Capstone: Your Own Dataset, Start to Finish

> *Seven chapters, seven separate pieces. This capstone is the only chapter with no fixed dataset — the reader's own choice, taken through every stage this course built one piece at a time. It's demonstrated here on the Palmer Penguins dataset, a real, small, freely available dataset used purely as a worked illustration — every step is written to be swapped for whatever dataset you actually pick.*

The Pipeline, and What It's Demonstrated On

Palmer Penguins — real physical measurements (bill length, bill depth, flipper length, body mass) of three penguin species from Palmer Station, Antarctica, plus island and sex. It's used here specifically because, unlike Chapter 7's own squeaky-clean Iris dataset, it has real missing values and a real categorical column — giving this capstone a genuine reason to exercise every stage of the pipeline, not just the modeling step.

Stage 1 — Collect

```
import pandas as pd

# swap this line for your own dataset:
#   - a downloaded CSV (as here)
#   - a scraped page (Chapter 1's own pattern)
#   - a live API, single or combined (Chapter 3's / Chapter 6's own pattern)
raw = pd.read_csv("penguins.csv")
print(raw.shape)
```

Stage 2 — Clean

```
def clean_penguins(df, report):
    before = len(df)
    df = df.dropna(subset=["species", "bill_length_mm"])
    report.append(f"Dropped {before - len(df)} row(s) missing species or bill_length_mm.")
```

```
    n_sex_missing = df["sex"].isna().sum()
    df["sex"] = df["sex"].fillna("unknown")
    report.append(f"Filled {n_sex_missing} missing sex value(s) with 'unknown'.")

    return df

report = [f"Started with {len(raw)} rows."]
cleaned = clean_penguins(raw, report)
report.append(f"Finished cleaning with {len(cleaned)} rows.")
```

Directly reusing Chapter 2's own shape: a function that cleans and appends plain-English lines to a shared `report` list, with the same kind of real per-column judgment call that chapter's own warn-box named — rows missing a core measurement are dropped, a missing category is labeled rather than guessed.

Stage 3 — Explore

```
report.append("--- Exploration ---")
report.append(str(cleaned.groupby("species")["bill_length_mm"].describe()[["mean", "std"]]))
report.append(f"Species counts: {cleaned['species'].value_counts().to_dict()}")
```

Chapter 5's own flowing, narrative style rather than a numbered checklist — a couple of well-chosen summaries, not an exhaustive re-run of every EDA step, since the real goal here is a working pipeline end to end, not a second full exploration chapter.

Stage 4 — Visualize

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(
    data=cleaned, x="bill_length_mm", y="flipper_length_mm",
    hue="species", palette=["#15803D", "#EAB308", "#4ade80"],
)
plt.title("Bill Length vs. Flipper Length by Species")
plt.savefig("capstone_dashboard.png")
```

Stage 5 — Model

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

features = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_mass_g"]
X = cleaned[features].dropna()
y = cleaned.loc[X.index, "species"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
accuracy = model.score(X_test, y_test)

report.append(f"--- Model ---")
report.append(f"KNN classifier accuracy on held-out test data: {accuracy:.2%}")
```

Chapter 7's own exact template — same algorithm, same honest framing (a light touch, not a substitute for `ml1`'s own full depth) — applied here to a dataset with genuinely more real-world texture than Iris had.

Stage 6 — Report

```
from pathlib import Path

Path("findings_report.txt").write_text("\n".join(report))
print("\n".join(report))
```

The same report-list pattern from Chapter 2, now spanning the entire pipeline rather than just the cleaning stage — one readable, saved artifact documenting exactly what happened to the data at every step, from raw file to trained model.

Chapter Attribution Table

STAGE	DRAWN DIRECTLY FROM
Collect	Chapter 1 (scraping), Chapter 3 (single live API), Chapter 6 (combining independent APIs) — swap freely depending on your own dataset's source
Clean	Chapter 2's own reusable, report-generating cleaning-function pattern
Explore	Chapter 5's own flowing, narrative EDA approach rather than a rigid checklist
Visualize	ds1-7/ds1-8's own chart-type choices, applied throughout Chapters 3-6
Model	Chapter 7's own light-touch KNN classifier template

STAGE	DRAWN DIRECTLY FROM
Report	Chapter 2's own report-list pattern, extended across the whole pipeline

SCOPE NOTE — WHAT THIS CAPSTONE DELIBERATELY DOESN'T DO

- No deep model tuning, cross-validation, or algorithm comparison — that's `ml1`'s own full-depth job, not this pipeline's.
- No text, image, or deep-learning-shaped project — `nlp1` / `nn1` / `llm1`-shaped work (like the sentiment-analyzer idea from this subject's own original roadmap brainstorm) belongs to the still-unfleshed `dsproj2` (Intermediate), not this Beginner course.
- No production deployment, scheduling, or serving — this pipeline runs once, by hand, producing local files.
- No guarantee your own chosen dataset will be as cooperative as Palmer Penguins — real data varies enormously, and adapting this template to genuine mess is expected, not a sign something went wrong.

One pipeline, swappable inputs

Every stage is written to be pointed at a different dataset with minimal changes.

The report as a real deliverable

A saved, readable summary of what happened — not just code that ran silently.

Every prior chapter, reused directly

Nothing new introduced here — this chapter's job is combination, not new material.

Honest scope, stated explicitly

What this pipeline can't do is named directly, not glossed over.

Extend This Project

Try these on your own:

- Pick a genuinely different real dataset — one you find yourself, downloaded or scraped — and run it through this exact pipeline, adapting each stage as needed.
- Turn `findings_report.txt` into a short Markdown file with the saved chart embedded, for a more presentable finished artifact.
- If your own dataset has a date column, add Chapter 4's own `.resample()` trend analysis as an extra stage.
- Once you've taken `ml1`, swap Stage 5's KNN classifier for a properly tuned model and compare the accuracy difference honestly.

Course Complete

That's all 8 projects of **Data Science & ML Projects (Beginner)** — a scraper, a cleaning pipeline, a live weather dashboard, a pandas-powered finance analyzer, a full EDA, a multi-source API aggregator, a first classifier, and this capstone — combining `ds1`'s own data-handling toolkit with one light, honest taste of `ml1`. The still-unfleshed `dsproj2` (Intermediate) remains as the final piece of the Data Science & ML subject's own six-course roadmap, and is exactly where NLP-shaped projects like a real sentiment analyzer belong.

CHAPTER 8 QUICK REFERENCE — COURSE SUMMARY

- **The full pipeline:** collect (1/3/6) → clean (2) → explore (5) → visualize (3-6) → model (7) → report (2, extended)
- Demonstrated on Palmer Penguins purely as a worked illustration — every stage is meant to be swapped for the reader's own real dataset
- The findings report is a genuine, saved deliverable, not just code that ran and printed to a terminal
- **Honest scope:** no deep model tuning, no NLP/deep-learning work (that's dsproj2's own job), no production deployment
- This completes the Data Science & ML Projects, Beginner course (8 chapters)