Appendix A

Python and Google Colab Quick Reference

Dr. Jose Mendoza, Academic Director and Clinical Associate Professor
Version 1.0 · July 2026

Appendix Information

PURPOSE

This appendix is a task-indexed reference for the Google Colab and Python actions that recur across the guide. It exists so that a student who already knows what the analysis requires does not have to search four chapters to remember how to perform the step. Every entry states the task, supplies a short runnable pattern, names the result that pattern should produce, and gives one check that confirms it did.

USE THIS APPENDIX WHEN

Consult this appendix when you know the analytic step you need and want the syntax for it: opening a notebook, loading a course file, inspecting a table, filtering, deriving a column, grouping, joining, reshaping, drawing a quick chart, fitting a chapter-specified model, verifying a transformation, diagnosing an error message, or preparing work for submission.

THIS APPENDIX DOES NOT

This appendix is a quick reference. It does not teach Python, does not teach statistical or analytical methods, does not reproduce the chapter labs, and does not supply answers to graded exercises. It catalogs only the commands the guide actually uses, not the full capability of pandas, Matplotlib, statsmodels, or scikit-learn. When an entry raises a question of judgment rather than syntax, it points to the chapter that owns the decision and stops there.

VERSION AND DATE

Version 1.0 · July 2026 · Language: English (United States)

SUGGESTED CITATION

Mendoza, J. (2026). Python and Google Colab quick reference. In *Applied business analytics for marketing decision-making: Business analytics and data visualization* (Appendix A, Version 1.0) [Open educational resource]. CC BY 4.0.

LICENSE AND RIGHTS

are used for identification only and do not imply endorsement. StyleCraft Collective is a fictional company created for instruction.

GENERATIVE AI USE

Generative artificial intelligence and other AI-assisted tools were used in the research, writing, revision, and production of this appendix, including outlining, preliminary drafts, revision of prose, support for code and analytical examples, and document formatting. These tools were used under the author's direction and are not credited as authors, researchers, or sources. The author determined the appendix's scope, boundaries, and content, and reviewed and approved all AI-assisted material: factual claims and citations were checked against the underlying sources rather than accepted from AI-generated summaries, and every code pattern was executed and its stated result confirmed. Responsibility for the accuracy, originality, and final form of this appendix rests entirely with the author. A fuller statement appears in the front matter of the complete guide.

COMPANION FILES

A runnable Appendix_A_Quick_Reference.ipynb built to pass Restart and run all, the small practice files and the larger synthetic demonstration files this appendix uses, a standalone command index, and the maintained package-version record are in the Applied Business Analytics companion repository.

How to Read an Entry

Entries in this appendix are organized by task, not by function name. You should be able to find a command by describing what you are trying to do, which is why Table A.1 and the command index in Section A.16 both begin with the task rather than the syntax.

Every entry has the same four parts. The code caption states the task. The code block supplies a short pattern. The italic line that follows names the result the pattern should produce — usually a shape or a type rather than a number, because your file is not the one this appendix was written against. The final line supplies one verification: a check that would fail if the step did something other than what you intended.

That fourth part is not decoration. The guide's second discipline is predict-then-verify (Section 1.7), and a reference that supplies syntax without a check trains the wrong habit. Where an entry transforms data, the verification is mandatory, not optional.

CONCEPT

Copy, Adapt, Verify

A pattern in this appendix is a starting point, not an answer. Copy it, replace every placeholder with the field names your file actually carries, run it, and then run the verification line. Code that executes without an error has proved only that it is valid Python. It has not proved that it computed what you meant.

The same rule governs code an AI assistant drafts for you. Appendix C supplies prompt templates for asking an assistant to produce any pattern in this appendix; Appendix D is where the exchange, the correction, and the verification are recorded.

Source: Course concept developed for this guide.

Placeholders appear in square brackets and capital letters. [FILE] stands for a file name, [COLUMN] for a column in your own table, [KEY] for a join key, and [WINDOW] for a date range you have declared. Replace all of them before running. A pattern that still contains a bracketed placeholder has not been adapted.

Most examples run against a small practice file, practice_transactions.csv, which ships with this appendix: twenty order lines, twelve orders, eight customers, and $1,312.00 of revenue, shaped exactly like StyleCraft's transactions table and small enough to check with a calculator. It is a practice file, not an assignment file. No number in this appendix answers a graded exercise. Sections A.11 and A.12 need more rows than twelve orders to be worth looking at, so the companion repository also ships synthetic demonstration files at scale; they are clearly labeled, they are not StyleCraft data, and no number in them means anything. Against your own file, expect your output to differ.

Table A.1

What to consult, by what you are trying to do

If you need to…	Go to
Open, run, save, or recover a notebook	A.1
Load a course file, or write a processed file back out	A.2
Remember a piece of Python syntax, or read an error message	A.3
Start a notebook with the right imports and an environment record	A.4
See what is in a table you have just loaded	A.5
Choose rows or columns, sort, or build a derived variable	A.6
Handle missing values, duplicates, wrong types, or dates	A.7

If you need to…	Go to
Summarize by group, or change the grain of a table	A.8
Combine two tables without corrupting either	A.9
Move between wide and long, or between transaction and customer grain	A.10
Draw a histogram, bar, line, scatter, or box plot	A.11
Split, fit, predict, score, or cross-validate	A.12
Prove that a step did what you intended	A.13
Diagnose an error or an implausible result	A.14
Export figures and tables and submit the work	A.15
Find a command when you cannot remember its name	A.16

Table A.2 records the StyleCraft tables this appendix loads and the grain of each, because almost every mistake in the sections that follow is a grain mistake wearing a different costume. One row of transactions is one order line, not one order and not one customer; a metric computed at the wrong grain is wrong even when every command in the cell is correct.

Table A.2

The StyleCraft tables this appendix refers to, and the grain of each

Table	One row is	Key
`transactions`	one order line	`order_line_id`
`customers`	one customer	`customer_id`
`products`	one product (SKU)	`product_id`
`stores`	one physical store	`store_id`
`campaigns`	one marketing campaign	`campaign_id`
`customer_features`	one customer, at a stated snapshot	`customer_id`
`marketing_daily`	one calendar day	`date`
`ab_test`	one randomized customer	`customer_id`
`transactions_enriched`	one order line, with dimensions joined on	`order_line_id`

A.1 Getting Started in Google Colab

Google Colab runs a Python notebook on a Google server rather than on your machine. Nothing needs to be installed. Two consequences follow from that, and both cause more lost work than any syntax error in this appendix: the machine you are given is temporary, and the variables you create live in that machine's memory rather than in the notebook file.

In other words, the notebook is saved; the runtime is not. Files you upload and variables you define disappear when the runtime is recycled, which happens after a period of inactivity and whenever you choose Restart. Everything in Section A.15 about running a notebook from top to bottom before submission follows from this one fact.

A.1.1 The First Five Minutes

Work through this list at the start of every session.

1. Open the notebook from the companion repository, from Brightspace, or from your own Google Drive. Opening from the repository gives you a read-only view; use File → Save a copy in Drive before you edit.
2. Confirm the notebook is connected. The Connect button at the top right becomes a RAM and Disk indicator once a runtime is attached.
3. Rename the file so it identifies you and the assignment, for example Assignment1_Mendoza.ipynb. Do not submit a file still called Copy of ....
4. Run the first cell and read its output. Code A.1 is that cell.
5. Load the data you need before you write any analysis, and check its shape (Section A.5) before you compute anything from it.

Code A.1. Confirm where you are and what you have

```
import sys
from pathlib import Path

print("Python:", sys.version.split()[0])
print("Working directory:", Path.cwd())

here = sorted(p.name for p in Path(".").iterdir() if p.is_file())
print(f"{len(here)} file(s) here:", here)
```

Expected result: the Python version, the working directory (usually /content in Colab), and a complete count and listing of the files the runtime can see.

Verify: the listing is complete rather than truncated, so a file absent from it really is absent — a truncated listing cannot support that conclusion. If the file you need is missing, Section A.2 shows how to put it there.

A.1.2 Cells, Running, and Stopping

A notebook is a sequence of cells. Code cells contain Python and produce output below themselves; text cells contain Markdown and carry your explanations. A cell's output shows only its last expression, which is why cells in this guide print results explicitly rather than relying on that behavior. Table A.3 lists the notebook actions you will use and what each one does to the runtime.

Table A.3

Notebook actions and what each one does

To do this	Use this	What happens
Run one cell	Shift+Enter, or the play button	Runs the cell and moves to the next
Run one cell and stay	Ctrl+Enter	Runs the cell in place
Run everything	Runtime → Run all	Runs every cell from the top in order
Stop a running cell	The stop button, or Runtime → Interrupt	Ends the cell; variables already created survive
Clear the memory	Runtime → Restart session	Every variable and imported module is lost; the notebook text is not
Start completely clean	Runtime → Restart and run all	The only honest test that a notebook reproduces
Clear the printed output	Edit → Clear all outputs	Removes results, keeps the code

Restarting matters more than it appears to. A notebook that runs correctly in the order you happened to execute the cells may fail completely when read from top to bottom, because a variable you defined in a cell you later deleted is still sitting in memory. Restart and run all is the check that catches this, and Section A.15 requires it before submission.

A.1.3 Saving and Downloading

Colab autosaves to Google Drive, but only after you have saved a copy there. Use File → Save a copy in Drive on any notebook you opened from a link. To hand work in, use File → Download and choose .ipynb, which preserves code, output, and text together. Download .py only when the course asks for a script, and produce a PDF only when the assignment asks for one; a PDF is a picture of the work, not the work.

A.2 Files and Paths

The guide uses one file workflow by default: read the course file straight from the companion location into a DataFrame, in one line, with the date columns parsed as dates. Uploading and mounting Drive are alternatives worth knowing, but they are not the default, because an uploaded file has to be uploaded again after every runtime restart, and a mounted Drive makes the notebook depend on one person's folder structure.

Code A.2. Read a course file from the companion location

```
import pandas as pd

# One line to change if the course files move. The current value is
# published on the companion resource page; do not retype the full
# address in every cell.
DATA = ("https://raw.githubusercontent.com/jrmst102/busin"
        "essanalytics/main/data/")

transactions = pd.read_csv(DATA + "practice_transactions.csv",
                           parse_dates=["order_date"])
print(transactions.shape)

# An Excel workbook is read the same way, naming the sheet.
# book = pd.read_excel(DATA + "[FILE].xlsx", sheet_name="[SHEET]")
```

Expected result: a tuple of (rows, columns). For the practice file it is (20, 13).

Verify: the row count matches what the file is documented to contain, and order_date is a date rather than text — confirm with transactions.dtypes (Section A.5).

Two arguments in that call are doing real work. The parse_dates list converts the named columns to true dates at read time, which is cheaper and safer than converting later; a date left as text sorts alphabetically and silently produces a wrong answer. And assigning the location to DATA once means a moved file is a one-line repair rather than a search through the notebook.

Code A.3. Upload a file from your own computer (Colab only)

```
from google.colab import files

uploaded = files.upload()          # opens a file chooser
print(list(uploaded.keys()))

import pandas as pd
transactions = pd.read_csv("[FILE].csv", parse_dates=["order_date"])
```

Expected result: a file chooser, then the name of each uploaded file.

Verify: the file appears in the listing produced by Code A.5. Remember that an uploaded file is lost on restart and has to be uploaded again.

Code A.4. Connect Google Drive (Colab only)

```
from google.colab import drive
drive.mount("/content/drive")

import pandas as pd
path = "/content/drive/MyDrive/[FOLDER]/[FILE].csv"
transactions = pd.read_csv(path, parse_dates=["order_date"])
```

Expected result: an authorization prompt, then the message Mounted at /content/drive.

Verify: the path exists before you read it. Use Code A.5 on the folder rather than guessing at the spelling of MyDrive.

Mount Drive when an assignment genuinely requires files that live in your own Drive. Do not mount it in a notebook you intend to submit for a group, because the path will not resolve on anyone else's account.

Code A.5. See what the runtime can actually find

```
from pathlib import Path

for p in sorted(Path(".").glob("*.csv")):
    print(p.name, f"{p.stat().st_size:,} bytes")
```

Expected result: one line per CSV file in the working directory, with its size. No output at all means there are no CSV files there.

Verify: the file you are about to load is on that list, spelled exactly as you will type it. File names are case-sensitive.

Code A.6. Write a processed file back out, and confirm it landed

```
orders = (transactions
          .groupby("order_id", as_index=False)["line_revenue"]
          .sum())

orders.to_csv("orders_summary.csv", index=False)

check = pd.read_csv("orders_summary.csv")
print("rows written:", len(check))
```

Expected result: the number of rows in the file you just wrote. For the practice file it is 12, one per order.

Verify: read the file back and compare its row count and its total to the frame you wrote. index=False keeps pandas from adding an unnamed index column that will confuse the next person who opens the file.

A file written into the working directory is on the temporary machine, not on yours. Download it with File → Download, or write it into a mounted Drive folder, before the runtime is recycled.

A.3 The Small Amount of Python You Need Repeatedly

This section is a memory aid for the handful of language features the guide's notebooks actually use. It is not an introduction to Python, and it deliberately omits classes, comprehensions beyond the simplest form, and everything else that a marketing analyst can be productive without.

Code A.7. Assignment, the basic types you will meet, and comments

```
store_id = "S09"                        # a string
unit_price = 88.00                      # a float (a decimal number)
quantity = 2                            # an integer (a whole number)
is_occasionwear = True                  # a Boolean: True or False
line_revenue = quantity * unit_price    # 176.0

metros = ["NYC Urban", "NYC Suburban", "Resort"]   # a list, ordered
thresholds = {"caution": 0.05, "stop": 0.10}       # a dictionary

print(line_revenue, metros[1], thresholds["stop"])
```

Expected result: 176.0 NYC Suburban 0.1.

Verify: list positions start at 0, so metros[1] is the second element. A dictionary is read by key, not by position.

The single equals sign assigns; the double equals sign compares. Anything after a # on a line is a comment, ignored by Python and read by the next person, who is usually you three weeks later. Indentation is not cosmetic: the four spaces at the start of a line are what tell Python which lines belong inside an if statement or a function.

Code A.8. Function calls, keyword arguments, comparisons, and if

```
n = len(transactions)                    # positional argument
total = round(transactions["line_revenue"].sum(), 2)

# A keyword argument is named at the call site. Both lines below
# do the same thing; the second says what the 2 means.
head_rows = transactions.head(3)
head_rows = transactions.head(n=3)

big = transactions["line_revenue"] > 100       # one answer per row
both = ((transactions["quantity"] >= 2)        # use & and |, not
        & (transactions["discount_pct"] == 0)) # "and" and "or"

if total > 1000 and n == 20:
    print(f"{n} lines totalling ${total:,.2f}")
else:
    print("this is not the file the cell expects")
```

Expected result: for the practice file, 20 lines totalling $1,312.00.

Verify: a keyword argument is matched by name, so its position does not matter and its meaning is visible. Named arguments recur throughout this appendix — parse_dates=, dropna=, validate=, random_state= — and reading them is how you know what a cell is doing. A

comparison against a column returns a column of True and False, one per row, not a single answer. Wrap each condition in parentheses and join them with & or |; Python's and and or do not work on columns.

The f before the quotation mark makes an f-string, which substitutes the value of anything in braces. The :,.2f inside the braces asks for thousands separators and two decimal places. Use it whenever you print money.

Code A.9. Write a rule once, in a function, and reuse it

```
def contribution(revenue, cost):
    """Contribution dollars for one line. Both arguments in dollars."""
    return revenue - cost

transactions["contribution"] = contribution(
    transactions["line_revenue"], transactions["line_cost"])

print(round(transactions["contribution"].sum(), 2))
```

Expected result: one number: total contribution dollars across the file.

Verify: the sum of the new column equals total revenue minus total cost. A rule written once in a function cannot drift between two cells, which is the whole argument for putting it there.

A.3.1 Reading an Error Message

Python error messages are read from the bottom upward. The last line names the error type and gives the specific complaint; that is the line to read first. The lines above it are the traceback, showing where the failure happened, and the topmost lines are usually inside the library rather than in your code.

Code A.10. Provoke an error deliberately and read what it says

```
try:
    transactions["revenue"].sum()   # the column is line_revenue
except KeyError as exc:
    print("KeyError:", exc)
    print("columns that do exist:", list(transactions.columns))
```

Expected result: KeyError: 'revenue', followed by the list of column names the table actually carries.

Verify: the printed list is the answer. A KeyError is almost always a spelling, a case difference, or a column that a previous cell renamed or dropped. Section A.14 catalogs the errors you will meet most.

A.4 Core Imports and the Reproducibility Block

Every notebook in this guide opens with the same three imports. Chapter-specific libraries are imported in the chapter that needs them, at the point of need, so that a reader can see which section introduced a dependency.

Code A.11. The canonical import block

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Expected result: no output. An import that succeeds is silent.

Verify: if any line raises ModuleNotFoundError, see Section A.14.5. The aliases np, pd, and plt are conventions, not requirements, but every example in this guide and in the companion notebooks assumes them.

Table A.4 names the libraries the guide depends on, what each is used for, and the chapter that introduces it. pandas supplies the DataFrame (McKinney, 2010), NumPy the numeric array beneath it (Harris et al., 2020), and Matplotlib the charts (Hunter, 2007); scikit-learn (Pedregosa et al., 2011) and statsmodels (Seabold & Perktold, 2010) arrive later, in Part II.

Table A.4

The libraries the guide uses, and the chapter that introduces each

Library	Used for	Introduced in
`pandas`	Tables: loading, cleaning, grouping, joining	Chapter 4
`numpy`	Numeric arrays and the isclose and allclose checks	Chapter 4
`matplotlib.pyplot`	Charts drawn in the notebook	Chapter 4
`scikit-learn`	Scaling, clustering, splits, pipelines, models, metrics	Chapters 6 and 8
`statsmodels`	Regression with interpretable coefficients; smoothing; proportion tests	Chapters 7, 10, and 11
`IPython.display`	Showing more than one result from a single cell	Chapter 4

Do not add imports the notebook does not use. An unused import is a small thing, but it is also a signal that code arrived from somewhere else without being read — which is exactly the habit this guide is built to prevent.

Code A.12. The environment record

```
import sys
import matplotlib
import sklearn
import statsmodels

print("Python      :", sys.version.split()[0])
print("pandas      :", pd.__version__)
print("numpy       :", np.__version__)
print("matplotlib  :", matplotlib.__version__)
print("scikit-learn:", sklearn.__version__)
print("statsmodels :", statsmodels.__version__)
```

Expected result: six version numbers, one for Python and one for each library the guide depends on.

Verify: compare these against the version record in the companion repository when a result you expect to reproduce does not. Place this cell near the top of any notebook you submit; it is the cheapest possible insurance against an unreproducible result.

Version numbers are not printed in this appendix's prose on purpose. Libraries change between terms, and a number frozen into a book ages badly. The maintained environment record in the companion repository is the authoritative statement of which versions the course notebooks were run against.

A.5 Inspecting a DataFrame

Run these five commands on every file, in this order, before computing anything from it. They take under a minute and they catch most of the failures that would otherwise appear much later, disguised as an analytical result.

Code A.13. Shape, column names, and the first rows

```
print("rows, columns:", transactions.shape)
print(list(transactions.columns))
print(transactions.head())
```

Expected result: a (rows, columns) tuple, the column names as a list, and the first five rows.

Verify: the row count matches what the file is documented to contain. A shape of (1, 13) usually means the delimiter was wrong; a shape with far more columns than expected usually means a stray comma in the source.

Code A.14. Types, structure, and summaries

```
from IPython.display import display

display(transactions.dtypes)
transactions.info()
display(transactions.select_dtypes(include="number").describe().round(2))
display(transactions.select_dtypes(exclude="number").describe())
```

Expected result: the type of each column; a structural summary with non-null counts; a numeric summary of count, mean, standard deviation, minimum, quartiles, and maximum; and, for the non-numeric columns, a count of distinct values with the most frequent one.

Verify: every column that should be a number is a number and every column that should be a date is a date. A price column reported as anything other than a numeric type — object or str, depending on the pandas version — is text, usually carrying currency symbols; see Code A.26.

A notebook renders only the last expression in a cell, so a cell listing four results shows one of them. The display function from IPython prints each one explicitly. Reading describe on the numeric columns is where implausible values first announce themselves: a maximum quantity of 999, a minimum price of zero, a mean far from the median.

Code A.15. Missing values, distinct values, and a frequency table

```
print(transactions.isna().sum())
print(transactions.nunique())
print(transactions["channel"].value_counts(dropna=False))
print(transactions["channel"].value_counts(normalize=True,
                                           dropna=False).round(3))
```

Expected result: a count of missing values per column, a count of distinct values per column, then the counts and the shares for one categorical column.

Verify: dropna=False is not optional. Without it, missing values vanish from the frequency table and the shares are computed on a denominator you did not choose. A category list longer than you expected usually means inconsistent labels; see Code A.21.

Table A.5 records what each of those commands is actually for. Read it as a set of questions rather than a set of functions: the question is what you have when you open an unfamiliar file.

Table A.5

What each inspection command answers

Command	The question it answers
`.shape`	How many rows and columns did I actually get?
`.columns`	What are the fields called, exactly?

Command	The question it answers
`.head() / .tail()`	What does a row look like, and does the end of the file look like the start?
`.dtypes`	Is each column stored as the type it should be?
`.info()`	Which columns have missing values, and how large is the table in memory?
`.describe()`	Are the numeric ranges plausible? Is the mean far from the median?
`.isna().sum()`	Where is data missing, and how much?
`.nunique()`	Which columns are keys, and which are categories?
`.value_counts()`	What labels exist in this category, and how common is each?

Code A.16. Row count and distinct-key check

```
print("rows                 :", len(transactions))
print("distinct order lines :", transactions["order_line_id"].nunique())
print("distinct orders      :", transactions["order_id"].nunique())
print("distinct customers   :", transactions["customer_id"].nunique())

assert transactions["order_line_id"].is_unique, "order_line_id is not a key"
```

Expected result: four counts, and silence from the assertion. For the practice file: 20 rows, 20 order lines, 12 orders, 8 customers.

Verify: the count of distinct order lines equals the row count. If it does not, the table has duplicates (Code A.25) and every total computed from it will be too high.

That last check is the one to run first on any fact table. It establishes the grain: what one row of this table represents. Chapter 3 owns that concept (Section 3.3); this appendix only supplies the command that confirms it.

A.6 Selecting, Filtering, Sorting, and Creating Columns

These six patterns account for most of the data handling in the guide's labs. Each of them produces a new object rather than changing the one you started from, which is deliberate: the original file stays available as evidence, and every step is reversible by rerunning the notebook from the top.

Code A.17. Select one column and several columns

```
revenue = transactions["line_revenue"]                    # a Series
keys = transactions[["order_id", "customer_id", "line_revenue"]]  # a frame

print(type(revenue).__name__, type(keys).__name__, keys.shape)
```

Expected result: Series DataFrame (20, 3).

Verify: single brackets return one column as a Series; double brackets return a table, even when the list has one name in it. Many puzzling errors later in a notebook come from having one where you meant the other.

Code A.18. Filter on one condition, several conditions, and a list

```
store_lines = transactions[transactions["channel"] == "Store"]

full_price_store = transactions[(transactions["channel"] == "Store")
                                & (transactions["discount_pct"] == 0)]

digital = transactions[transactions["channel"].isin(["Online", "App"])]

print(len(store_lines), len(full_price_store), len(digital))
```

Expected result: three row counts. Each is a subset, so each is no larger than the original.

Verify: the counts of mutually exclusive, exhaustive subsets add back to the original row count. If they do not, a category label is missing from your isin list, or a row has a missing value in the filtered column — which is excluded by every comparison, including inequalities.

Code A.19. Sort, with a tie-break that makes the order reproducible

```
# Sorted on the measure alone: ties fall wherever the file put them.
top = transactions.sort_values("line_revenue", ascending=False)

# Sorted with an identifier as the tie-break: reproducible anywhere.
stable = transactions.sort_values(["line_revenue", "order_line_id"],
                                  ascending=[False, True])

print(stable[["order_line_id", "line_revenue"]].head(3))
```

Expected result: the three largest lines from the reproducibly sorted frame, largest first.

Verify: print stable rather than top. Sorting on the measure alone leaves ties in whatever order the file happened to arrive in, so a ranked list can differ between machines. Adding the identifier as a second sort key makes the ordering reproducible by anyone.

Code A.20. Create a derived variable from an explicit rule

```
transactions["gross_line_revenue"] = (transactions["quantity"]
                                      * transactions["unit_price"])
transactions["discount_dollars"] = (transactions["gross_line_revenue"]
                                    - transactions["line_revenue"])
transactions["is_discounted"] = transactions["discount_pct"] > 0

print(transactions.loc[transactions["is_discounted"],
                       ["order_line_id", "gross_line_revenue",
                        "discount_dollars", "line_revenue"]])
```

Expected result: one row per discounted line, showing the arithmetic.

Verify: gross revenue minus discount dollars equals line revenue on every row. Check it with np.allclose rather than by eye: np.allclose(transactions["gross_line_revenue"] - transactions["discount_dollars"], transactions["line_revenue"]).

A derived variable is a definition, and a definition belongs in writing. When the derived column is a named metric — average order value, repeat purchase rate, contribution margin — the numerator, denominator, window, filters, and grain are cataloged in Appendix B, and the code here should implement that entry rather than inventing a second version of it.

Code A.21. Rename a column and recode a category

```
# Rename onto a NEW name. Rebinding transactions here would break
# every later cell that still refers to line_revenue.
renamed = transactions.rename(columns={"line_revenue": "revenue"})
print(list(renamed.columns)[-3:])

channel_map = {"Online": "Digital", "App": "Digital", "Store": "Store"}
transactions["channel_group"] = transactions["channel"].map(channel_map)

print(transactions["channel_group"].value_counts(dropna=False))
assert transactions["channel_group"].notna().all(), "an unmapped label"
```

Expected result: the last three column names of the renamed copy, then the counts for each new group, and silence from the assertion.

Verify: map turns any label missing from the dictionary into a missing value, silently. The assertion is what turns that silence into a message. List the distinct labels with value_counts before writing the dictionary, not after.

Code A.22. Assign to a copy, never to a slice of a slice

```
store = transactions[transactions["channel"] == "Store"].copy()
store["revenue_per_unit"] = store["line_revenue"] / store["quantity"]

print(store["revenue_per_unit"].round(2).head())
```

Expected result: one revenue-per-unit figure per store line.

Verify: the new column exists on store and not on transactions. Without .copy(), pandas may warn about setting a value on a copy of a slice, and in recent versions the assignment simply does not reach the original table. Filter, copy, then assign — in that order.

A.7 Missing Values, Duplicates, Types, and Dates

This section supplies the commands. It does not supply the decisions. Whether a missing discount means no discount or an unrecorded one, whether an outlier is an error or a legitimate extreme, and whether a near-duplicate is a re-export or a genuine second order are all analytic questions that Chapter 4 owns and that a command cannot answer.

CONCEPT

A Command Is Not a Justification

Every entry in this section changes the data. Running one of them records nothing about why it was appropriate. The remedy you choose, the rows it touched, and the bias it introduces belong in the verification log Chapter 4 requires, and — when an AI assistant proposed the remedy — in the record Appendix D requires.

The practical test is simple. If someone asked why the file now has forty fewer rows than the file you were given, could you answer without rerunning anything?

Source: Course concept developed for this guide.

Code A.23. Find the missing values and size the problem

```
missing = (transactions.isna().sum().rename("missing").to_frame()
           .assign(share=lambda d: (d["missing"] / len(transactions))
                   .round(4)))

print(missing[missing["missing"] > 0])
```

Expected result: one row per column that has any missing values, with the count and the share of the file. No output means nothing is missing.

Verify: compare the columns that appear against the data dictionary. A column documented as always present but missing in 3% of rows is a data problem to trace, not a value to impute.

Code A.24. Fill or drop, only after the decision has been made

```
before = len(transactions)

# Decision recorded in the verification log: an absent discount on a
# full-price line means no discount was applied.
transactions["discount_pct"] = transactions["discount_pct"].fillna(0.0)

# Decision: a line with no quantity cannot be valued, so it is excluded
# from revenue work and counted in the log.
computable = transactions.dropna(subset=["quantity"])

print("rows:", before, "| rows with a usable quantity:", len(computable))
```

Expected result: two counts. The difference between them is the number of rows the decision removed.

Verify: print the row count before and after, every time, and record both. A fill that changes a total and a drop that changes a row count are the two most common sources of a number that no one can reconcile later.

Code A.25. Exact duplicates and duplicates on the identity columns

```
exact = int(transactions.duplicated().sum())

identity = ["order_id", "customer_id", "product_id",
            "quantity", "unit_price"]
near = int(transactions.duplicated(subset=identity).sum())

print("exact duplicates:", exact,
      "| duplicates on the identity columns:", near)

deduped = transactions.drop_duplicates().copy()
```

Expected result: two counts. The second is at least as large as the first.

Verify: row count before minus exact duplicates equals row count after. A near-duplicate — the same order re-exported with a different timestamp — is invisible to the first check and visible to the second, which is why both are run.

Code A.26. Coerce text to a number, and a count to a nullable integer

```
# Run this against the raw file, where the defects actually live.
raw = pd.read_csv(DATA + "[RAW FILE].csv")
before_na = int(raw["unit_price"].isna().sum())

price_text = raw["unit_price"].astype("string")
raw["unit_price"] = pd.to_numeric(
    price_text.str.replace(r"[$,]", "", regex=True), errors="coerce")

after_na = int(raw["unit_price"].isna().sum())

# A count that can legitimately be missing needs a nullable
# integer type; plain int cannot hold one.
raw["quantity"] = raw["quantity"].astype("Int64")

print(before_na, "->", after_na,
      "| quantity dtype:", raw["quantity"].dtype)
```

Expected result: two missing-value counts, and Int64 with a capital I as the new type.

Verify: the two counts must be equal. errors="coerce" turns anything it cannot parse into a missing value rather than raising, so a rise in the count is the coercion telling you it failed on rows you have not looked at. Find them before continuing.

Table A.6 pairs the type symptoms you will actually see with the coercion that repairs each. It lists commands, not verdicts: whether a given value in a given file should be repaired, excluded, or traced back to its source is Chapter 4's question, and the defect catalog that governs the StyleCraft raw file is Chapter 4's too.

Table A.6

Type problems and the coercion that repairs each

Symptom	Cause	Repair
A price column is not a numeric dtype	Currency symbols or thousands separators in the text	Strip with .str.replace, then pd.to_numeric
Dates sort alphabetically	The column is still text	pd.to_datetime with errors="coerce"
A count column will not accept a missing value	Plain int cannot hold one	Cast to the nullable Int64
A 0/1 flag behaves like text	It was read as "True"/"False" strings	Map explicitly, then .astype(int)
A category has more labels than expected	Case and spelling variants	Standardize with .str.strip().str.title(), then .map

Code A.27. Parse dates with explicit error handling, then read the calendar

```
transactions["order_date"] = pd.to_datetime(
    transactions["order_date"], errors="coerce")
unparsed = int(transactions["order_date"].isna().sum())

transactions["order_year"] = transactions["order_date"].dt.year
transactions["order_month"] = transactions["order_date"].dt.to_period("M")
transactions["order_weekday"] = transactions["order_date"].dt.day_name()

print("dates that would not parse:", unparsed)
assert unparsed == 0, "investigate every unparsed date before continuing"
```

Expected result: a count of unparsed dates, and silence from the assertion when that count is zero.

Verify: unparsed dates are not a nuisance to be dropped. A date that fails to parse is usually a different format in part of the file, which means part of the file came from somewhere else.

Code A.28. Check implausible values with Boolean conditions

```
off_catalog = transactions[(transactions["unit_price"] < 24)
                           | (transactions["unit_price"] > 90)]
big_baskets = transactions[transactions["quantity"] > 20]
negative = transactions[transactions["line_revenue"] < 0]

print("lines priced outside the catalog band:", len(off_catalog))
print("lines with an implausible quantity   :", len(big_baskets))
print("lines with negative revenue          :", len(negative))
```

Expected result: three counts, each ideally zero. Any non-zero count is a list of rows to inspect, not a list of rows to delete.

Verify: the bounds come from the business, not from the data. StyleCraft's catalog runs from $24 to $90, so a $4.99 line is a defect even though it is a perfectly valid number. Write the bound you expect before you look at what the file contains.

A.8 Grouping and Aggregation

Grouping changes the grain of a table, and a change of grain changes what a row means. Lead with the grain, not with the syntax: state what one input row represents, what one output row should represent, and which total must survive the change. Then write the groupby.

> **CONCEPT**
>
> **Every Aggregation Has Two Grains and One Invariant**
>
> Before running any groupby, write four things down: the input grain, the output grain, the aggregation rule for each column, and the total or count that must be identical on both sides.
>
> Rolling order lines up to orders should preserve total revenue exactly. Rolling orders up to customers should preserve both total revenue and the count of orders. If an invariant you named does not hold, the aggregation is wrong — however sensible the output table looks.
>
> Source: Course concept developed for this guide.

Two arguments appear on every groupby in this guide. Pass dropna=False so that rows with a missing group key are visible rather than discarded, and pass observed=True whenever a grouping key is or could become a categorical, so that combinations which never occurred do not appear as empty rows and quietly change a denominator. Treat both as house rules rather than as options.

Code A.29. Group by one key, with named aggregations

```
# Input grain: one order line.  Output grain: one channel.
# Rule: lines counted, orders counted distinctly, units and
# revenue summed, revenue median reported alongside the mean.
# Invariant: summed revenue equals the file's total revenue.
by_channel = (transactions
              .groupby("channel", as_index=False,
                       dropna=False, observed=True)
              .agg(lines=("order_line_id", "size"),
                   orders=("order_id", "nunique"),
                   units=("quantity", "sum"),
                   revenue=("line_revenue", "sum"),
                   median_line=("line_revenue", "median")))

print(by_channel)
```

Expected result: one row per channel, with five columns you named yourself.

Verify: the revenue column sums to the file's total revenue. Named aggregation — new_name=("column", "function") — is the house style because it produces readable column names instead of a multi-level index nobody can index into.

The two keyword arguments are deliberate. as_index=False returns a flat table rather than one indexed by the grouping key, which is what you want when the result feeds a merge or a chart. And dropna=False makes the treatment of missing group keys explicit; without it pandas drops those rows without comment, and the shares you compute in Code A.31 will be shares of a denominator you did not choose.

Code A.30. Group by several keys and return a flat table

```
# Input grain: one order line.  Output grain: one month-channel
# pair.  Invariant: total revenue is unchanged.
transactions["order_month"] = (transactions["order_date"]
                               .dt.to_period("M"))

grid = (transactions
        .groupby(["order_month", "channel"],
                 dropna=False, observed=True)
        .agg(revenue=("line_revenue", "sum"))
        .reset_index())

print(grid.head())
```

Expected result: one row per month-and-channel combination that appears in the data.

Verify: the revenue total is unchanged from the ungrouped file. Use reset_index() when a groupby without as_index=False has left the keys in the index; observed=True keeps categorical keys from generating empty combinations that never occurred.

Code A.31. Counts, rates, and shares computed from pooled totals

```
# Input grain: one order line.  Output grain: one channel.
# revenue_per_order is an ORDER-level measure, so its denominator
# is a distinct order count, never a row count.
by_channel = (transactions
              .groupby("channel", dropna=False, observed=True)
              .agg(orders=("order_id", "nunique"),
                   customers=("customer_id", "nunique"),
                   revenue=("line_revenue", "sum")))

by_channel["revenue_share"] = (by_channel["revenue"]
                               / by_channel["revenue"].sum())
by_channel["revenue_per_order"] = (by_channel["revenue"]
                                   / by_channel["orders"])

print(by_channel.round(3))
```

Expected result: one row per channel with three counts and two computed measures.

Verify: the shares sum to 1. Compute every rate from pooled numerators and denominators, as here. Averaging a rate that is already a rate gives each group equal weight regardless of size, which is a different — and almost always wrong — number.

A distinct count is not a row count. nunique on order_id counts orders; size counts order lines. Choosing the wrong one is the most common way an order-level metric ends up computed at line grain, and the resulting average order value is always too low. Appendix B's entry for average order value states the grain explicitly for exactly this reason.

Code A.32. Change the grain, and prove the change preserved the total

```
# Input grain: one order line.  Output grain: one order.
# Invariant: total revenue, and the count of distinct orders.
orders = (transactions
          .groupby("order_id", as_index=False, dropna=False)
          .agg(customer_id=("customer_id", "first"),
               order_date=("order_date", "min"),
               lines=("order_line_id", "size"),
               units=("quantity", "sum"),
               order_revenue=("line_revenue", "sum")))

assert len(orders) == transactions["order_id"].nunique()
assert np.isclose(orders["order_revenue"].sum(),
                  transactions["line_revenue"].sum())
assert (transactions.groupby("order_id")["customer_id"]
        .nunique().eq(1).all()), "an order carries more than one customer"

print("orders:", len(orders),
      "| AOV:", round(orders["order_revenue"].mean(), 2))
```

Expected result: the order count and the average order value. For the practice file: 12 orders, AOV $109.33.

Verify: three assertions, and each catches a different failure. The first catches a grain that is not what you declared. The second catches revenue lost or duplicated in the roll-up. The third catches the quiet danger in "first": it is honest only when the attribute is constant within the group.

Table A.7 records those four statements for that one example, and is the form to copy into a verification log before every aggregation you write.

Table A.7

The four things to record before every aggregation

Record	Practice-file example
Input grain	One order line
Output grain	One order

Record	Practice-file example
Aggregation rule per column	Revenue and units summed; date minimized; customer taken as first
Invariant	Total revenue is $1,312.00 before and after

A.9 Joining Tables

Joins deserve more verification than any other operation in this appendix, because a join failure is both easy to cause and hard to see. A join that quietly duplicates rows raises every total in the notebook, and every downstream number remains internally consistent while being wrong.

> **CONCEPT**
>
> **Declare the Cardinality, Then Let the Code Enforce It**
>
> Before writing a merge, say aloud what the relationship is: many transaction lines to one store, many orders to one customer, one campaign to one performance row. Then pass that statement to the validate argument so pandas refuses the join if the data disagrees.
>
> Bracket the merge with a row count before and an assertion after. A left join that changes the row count has not enriched the table; it has multiplied it.
>
> Source: Course concept developed for this guide.

Code A.33. The validated merge

```
# Enrich onto a NEW name. Rebinding transactions here would mean
# the next cell that merges onto it collides with these columns.
rows_before = len(transactions)

enriched = transactions.merge(
    stores[["store_id", "store_name", "store_type"]],
    on="store_id",
    how="left",
    validate="many_to_one",
    indicator=True)

assert len(enriched) == rows_before, "the join changed the row count"
print(enriched["_merge"].value_counts())
print("lines with no store row:",
      int(enriched["store_type"].isna().sum()))

enriched = enriched.drop(columns="_merge")
```

Expected result: a breakdown of matched and unmatched rows, then a count of lines that found no store. For the practice file that count is 10 — the lines whose store_id is the reserved value ONLINE.

Verify: the row count is unchanged and the unmatched count is a number you can explain. An unmatched count you cannot explain is a data problem, not a rounding detail. Drop the indicator column once you have read it; a verification artifact does not belong in a certified table. enriched is the line-grain table the rest of this appendix uses when it needs store attributes, and Code A.40 rolls it up for the charts.

Three arguments carry the weight. The on argument names the key explicitly rather than letting pandas guess from matching column names. The validate argument states the expected cardinality and raises a MergeError when the data violates it. And indicator=True adds a temporary _merge column recording, for each row, whether it matched — which is how the unmatched rows become visible instead of becoming missing values.

Code A.34. Inspect the unmatched keys instead of ignoring them

```
unmatched = sorted(set(transactions["store_id"])
                   - set(stores["store_id"]))

print("store_id values with no row in stores:", unmatched)
print("lines affected:",
      int(transactions["store_id"].isin(unmatched).sum()))
```

Expected result: the list of key values present in the left table and absent from the right, and the number of rows they account for.

Verify: every value on that list is either a documented reserved value — ONLINE and APP in StyleCraft's transactions — or a defect. If you cannot classify it into one of those two categories, do not proceed.

Code A.35. Refuse a many-to-many join rather than discover it later

```
# A right-hand table must carry at most one row per key for a
# many-to-one join to be honest. This one carries two.
right = pd.concat([products[["product_id", "unit_price"]]] * 2)

try:
    transactions.merge(right, on="product_id", how="left",
                       validate="many_to_one")
except pd.errors.MergeError as exc:
    print("MergeError:", exc)
    print("duplicate keys on the right:",
          int(right["product_id"].duplicated().sum()))
```

Expected result: a MergeError naming the violated cardinality, and a count of duplicated keys in the right-hand table.

Verify: this is the error you want. Without validate, the same join would have succeeded and returned a table with more rows than it started with. Fix the right-hand table — usually by drop_duplicates(subset=["[KEY]"]) after establishing which duplicate is correct — rather than removing the validate argument.

Table A.8 sets out the four join types and the circumstances in which each is defensible. The guide's default is the left join, because a fact table enriched with dimension attributes should keep every fact.

Table A.8

Join types and when each is justified

how=	Keeps	Use it when
"left"	Every row of the left table	Enriching a fact table with dimension attributes. The default choice in this guide.
"inner"	Only rows that match on both sides	The analysis is defined only for matched records, and you have counted what the match dropped.
"outer"	Every row of both tables	Reconciling two sources against each other. Rare in the labs; always followed by an unmatched-row report.
"right"	Every row of the right table	Almost never. Swap the tables and use a left join instead, which is easier to read.

A.10 Reshaping Data

Reshaping is another grain question. Before reaching for a function, say what one row of the result should represent. Wide layouts — one row per group, one column per category — read well on a page. Long layouts — one row per observation — are what charts and models expect.

Code A.36. pivot_table: one row per group, one column per category

```
# Input grain: one order line.  Output grain: one month, with one
# column per channel.  Invariant: the grand total is unchanged.
wide = pd.pivot_table(transactions,
                      index="order_month",
                      columns="channel",
                      values="line_revenue",
                      aggfunc="sum",
                      fill_value=0.0,
                      observed=True)

print(wide.round(2))
```

Expected result: a table with one row per month and one column per channel.

Verify: the grand total of the wide table equals the total of the source column. Set fill_value explicitly: an absent combination means no revenue, and leaving it as a missing value will break the arithmetic in the next cell.

Code A.37. crosstab: counts and row shares

```
# lines is the line-grain table with customer attributes joined on,
# built with the validated merge of Code A.33.
lines = transactions.merge(customers[["customer_id", "home_metro"]],
                           on="customer_id", how="left",
                           validate="many_to_one")

counts = pd.crosstab(lines["home_metro"], lines["channel"])
shares = pd.crosstab(lines["home_metro"], lines["channel"],
                     normalize="index")

print(counts)
print(shares.round(3))
```

Expected result: a table of counts, then the same table as row proportions.

Verify: each row of the normalized table sums to 1. Choose the direction deliberately: normalize="index" answers "of this metro's lines, what share were online?" and normalize="columns" answers "of online lines, what share came from this metro?" They are different questions with different answers.

Code A.38. melt: turn a wide table back into one row per observation

```
long = (wide.reset_index()
        .melt(id_vars="order_month",
              var_name="channel",
              value_name="revenue"))

print(long.head())
```

Expected result: one row per month-and-channel combination, with the value in a single column named revenue.

Verify: the total of the value column equals the grand total of the wide table, and the row count equals rows times columns of the wide table. Charts and models want this shape; a page wants the wide one.

Code A.39. Convert transaction grain to customer grain

```
# Input grain: one order line.  Output grain: one purchasing
# customer, through an order-grain table in between.  Invariants:
# distinct customers, distinct orders, and total revenue.
analysis_date = pd.Timestamp("[ANALYSIS DATE]")

orders = (transactions
          .groupby(["order_id", "customer_id"], as_index=False)
          .agg(order_revenue=("line_revenue", "sum"),
               order_date=("order_date", "min")))

cf = (orders.groupby("customer_id")
      .agg(frequency=("order_id", "nunique"),
           monetary=("order_revenue", "sum"),
           last_order=("order_date", "max"))
      .assign(recency_days=lambda d: (analysis_date
                                      - d["last_order"]).dt.days,
              aov=lambda d: (d["monetary"] / d["frequency"]).round(2))
      .drop(columns="last_order"))

assert len(cf) == transactions["customer_id"].nunique()
assert cf["frequency"].sum() == transactions["order_id"].nunique()
assert np.isclose(cf["monetary"].sum(), transactions["line_revenue"].sum())
assert (cf["recency_days"] >= 0).all()
```

Expected result: one row per customer who purchased, carrying frequency, monetary value, recency in days, and average order value. Four assertions pass silently.

Verify: the four assertions are the point of the cell. Customers equal distinct customers; summed frequency equals distinct orders; summed monetary value equals total revenue; and no recency is negative, which would mean an order dated after the analysis date.

Two steps, not one, and the order matters. Rolling lines straight to customers in a single groupby makes it easy to count order lines where you meant to count orders. Going through an order-grain table first makes each grain change small enough to verify. The customer feature definitions themselves — including the exposure denominator that turns counts into rates — belong to Chapter 6; this pattern is the mechanics only.

A.11 Basic Charts in Python

These are working charts: quick, plain, and made to be looked at by the analyst rather than by an executive. Chapters 12 and 13 govern chart selection, encoding, visual integrity, and communication, and Appendix E is the checklist any chart must pass before it leaves your workspace. Nothing in this section overrides any of that.

Every pattern here uses the object-oriented form — fig, ax = plt.subplots() — rather than the shorter pyplot calls, because the axes object is what you need in order to label anything. And every chart in this guide carries axis labels with units and a title that says what is being shown. An unlabeled chart is not a fast chart; it is an unreadable one.

Every chart in this section is drawn from one frame, order_view: one row per order, carrying order_revenue, units, order_date, and the store_type the order belongs to. Code A.40 builds it, with the validated merge of Code A.33 followed by the grain change of Code A.32; the six entries after it reuse it without rebuilding it. The name is deliberately not orders — Code A.32's orders frame has no store_type, and reusing one name for two different tables is how a notebook starts producing numbers nobody can trace.

> **CONCEPT**
>
> **Aggregate First, Then Plot**
>
> Most misleading charts in student work are misleading before the plotting call. The chart draws exactly what it was handed; the error is in what was handed to it.
>
> So state the grain of the frame you are plotting, aggregate to that grain deliberately, and print the number of rows the chart excludes as missing. A chart that silently drops rows is the visual form of a silent row drop.
>
> Source: Course concept developed for this guide.

That rule applies to every entry below, including the ones whose code does not repeat it. mean, sum, and scatter all skip missing values without saying so, so print the excluded count beside any chart built from a column that can be missing — Code A.41 shows the line, and it is two lines of work in every other entry.

Code A.40. Build the chart frame once, and reconcile it

```
# Every chart below is drawn from order_view. Build it once, from
# the enriched line-grain table of Code A.33: label the digital
# orders rather than leaving them missing, then roll up to orders.
enriched["store_type"] = enriched["store_type"].fillna("Digital")

order_view = (enriched
              .groupby("order_id", as_index=False,
                       dropna=False, observed=True)
              .agg(store_type=("store_type", "first"),
                   order_date=("order_date", "min"),
                   units=("quantity", "sum"),
                   order_revenue=("line_revenue", "sum")))

assert len(order_view) == transactions["order_id"].nunique()
assert np.isclose(order_view["order_revenue"].sum(),
                  transactions["line_revenue"].sum())
print("orders:", len(order_view), "| revenue:",
      round(order_view["order_revenue"].sum(), 2))
```

Expected result: the order count and the revenue total, with both assertions silent. For the practice file: 12 orders and $1,312.00.

Verify: the grain change preserved the order count and the revenue total. Label the digital orders explicitly instead of leaving store_type missing: a missing group key is dropped by most

charting code without comment, and the chart then answers a narrower question than its title claims.

Code A.41. Histogram of one distribution

```
revenue = order_view["order_revenue"].dropna()
print("orders charted:", len(revenue),
      "| orders excluded as missing:", len(order_view) - len(revenue))

fig, ax = plt.subplots(figsize=(7, 4))
ax.hist(revenue, bins=40, edgecolor="white")
ax.set_xlabel("Order revenue ($)")
ax.set_ylabel("Number of orders")
ax.set_title("Distribution of order revenue")
fig.tight_layout()
plt.show()
```

Expected result: a histogram, preceded by a count of what was charted and what was excluded.

Verify: the counts printed above the chart account for every row. Vary the bin count and look again: a shape that appears at 40 bins and disappears at 20 is a property of the bins, not of the data.

Code A.42. Bar chart from a table you aggregated first

```
by_type = (order_view.groupby("store_type")["order_revenue"]
           .mean().sort_values(ascending=False))

fig, ax = plt.subplots(figsize=(7, 4))
ax.bar(by_type.index, by_type.to_numpy())
ax.set_xlabel("Store type")
ax.set_ylabel("Average order value ($)")
ax.set_title("Average order value by store type")
fig.tight_layout()
plt.show()
```

Expected result: one bar per store type, tallest first.

Verify: the bars begin at zero, which matplotlib does by default for bar charts and which Appendix E treats as a stop-ship requirement for ordinary magnitudes. Sorting by the measure is a choice; sorting alphabetically hides the comparison the chart exists to make.

Code A.43. Line chart of an ordered series

```
weekly = (marketing_daily.set_index("date")["revenue"]
          .resample("W").sum())

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(weekly.index, weekly.to_numpy(), linewidth=0.9)
ax.set_xlabel("Week ending")
ax.set_ylabel("Revenue ($)")
ax.set_title("Weekly revenue")
fig.tight_layout()
plt.show()
```

Expected result: a single line across the whole window.

Verify: the index is sorted and has no gaps: weekly.index.is_monotonic_increasing should be True. A line chart drawn from unsorted dates produces a scribble that looks like volatility.

Code A.44. Scatterplot of two measures

```
fig, ax = plt.subplots(figsize=(7, 4))
ax.scatter(order_view["units"], order_view["order_revenue"], s=8, alpha=0.3)
ax.set_xlabel("Units in the order")
ax.set_ylabel("Order revenue ($)")
ax.set_title("Order revenue against basket size")
fig.tight_layout()
plt.show()
```

Expected result: one point per order.

Verify: one mark is one order, not one line and not one customer. With thousands of points, s and alpha control overplotting; without them a dense cloud hides its own shape.

Code A.45. Box plot by group

```
labels, groups = [], []
for key, part in order_view.groupby("store_type", dropna=False,
                                    observed=True):
    values = part["order_revenue"].dropna()
    print(f"{key}: charting {len(values)}, excluding "
          f"{len(part) - len(values)} as missing")
    labels.append(str(key))
    groups.append(values.to_numpy())

fig, ax = plt.subplots(figsize=(7, 4))
ax.boxplot(groups, whis=1.5)
ax.set_xticks(range(1, len(labels) + 1))
ax.set_xticklabels(labels)
ax.set_ylabel("Order revenue ($)")
ax.set_title("Order revenue by store type")
fig.tight_layout()
plt.show()
```

Expected result: an exclusion count per group, then one box per group in the order the groups appear in labels.

Verify: the tick labels line up with the boxes. Set them explicitly, as here; a box plot whose categories are labeled by position is one reordering away from being wrong.

Code A.46. Add a labeled reference line

```
overall = order_view["order_revenue"].mean()

fig, ax = plt.subplots(figsize=(7, 4))
ax.bar(by_type.index, by_type.to_numpy())
ax.axhline(overall, linestyle="--", linewidth=0.9, color="0.3",
           label=f"All orders: ${overall:,.0f}")
ax.set_xlabel("Store type")
ax.set_ylabel("Average order value ($)")
ax.set_title("Average order value by store type")
ax.legend(fontsize=8)
fig.tight_layout()
plt.show()
```

Expected result: the same bars with a dashed horizontal line and a legend entry naming its value.

Verify: the reference line is labeled with what it represents. An unlabeled line is an unexplained claim, and Appendix E treats it as one. Use axvline for a vertical reference, such as the date a store opened.

Code A.47. Small multiples on shared axes

```
types = sorted(order_view["store_type"].unique())

fig, axes = plt.subplots(1, len(types),
                         figsize=(3.2 * len(types), 3.2),
                         sharex=True, sharey=True)

axes = np.atleast_1d(axes)     # one group still returns a bare Axes

for ax, t in zip(axes, types):
    part = order_view[order_view["store_type"] == t]
    values = part["order_revenue"].dropna()
    print(f"{t}: charting {len(values)}, excluding "
          f"{len(part) - len(values)} as missing")
    ax.hist(values, bins=25, edgecolor="white")
    ax.set_title(f"{t}\nn = {len(values):,}", fontsize=9)
    ax.set_xlabel("Order revenue ($)")

axes[0].set_ylabel("Number of orders")
fig.tight_layout()
plt.show()
```

Expected result: one panel per group, side by side, on identical axes.

Verify: sharex=True and sharey=True are mandatory, not stylistic. Panels on different scales invite exactly the comparison they make invalid. Printing n in each title is what keeps a five-order panel from looking as authoritative as a five-thousand-order one.

Table A.9

Chart command by the question you brought to the data

Question	Command	Entry
How is one measure distributed?	ax.hist, ax.boxplot	A.41, A.45
How do groups compare on one measure?	ax.bar	A.42
How does a measure move over time?	ax.plot	A.43
How do two measures move together?	ax.scatter	A.44
How does this compare with a target or a baseline?	ax.axhline	A.46
Does the pattern hold within every group?	plt.subplots(1, n)	A.46

Table A.9 is a command lookup, and only that. Which form a question actually calls for is not settled here. The reasons these forms work — the perceptual hierarchy that puts position and length above area and angle, the conditions under which a composition chart misleads, the mechanics that make a chart dishonest — are Chapter 12's, and the sign-off any chart must pass is Appendix E's. Consult them before choosing; consult Table A.9 only once the choice is made.

A.12 Common Modeling Workflow Syntax

This section is syntax support and nothing else. It covers the handful of modeling calls that recur across Chapters 7 through 11, so that a student who has already decided what to fit does not have to reopen four chapters to remember the argument order. Every choice these patterns leave open — which features, which model, which threshold, what the coefficients mean — belongs to the chapter named beside it.

CONCEPT

Syntax Here, Interpretation There

A model that runs is not a model that is right. Nothing in this section tells you whether the split respected the unit of prediction, whether a feature leaks the outcome, whether the threshold reflects the real cost of being wrong, or whether the coefficient can bear the verb you want to attach to it.

Chapter 7 owns regression interpretation, Chapter 8 owns the predictive frame and leakage, Chapter 9 owns classification and thresholds, Chapter 10 owns forecasting, and Chapter 11 owns causal claims. Use this section to write the cell; use those chapters to defend it.

Source: Course concept developed for this guide.

Every pattern in this section assumes one frame, model_df: the customer-grain modeling table the chapter builds, indexed by customer_id, carrying the customer_features columns defined in Chapter 6 together with the label the chapter defines — churned for a classification target, spend_next6m for a numeric one. Building that frame is Chapter 8's work, not this appendix's, because the snapshot date, the feature window, the outcome window, and the eligibility rule are analytic decisions rather than syntax. The index matters: every entry below relies on model_df being keyed by customer_id rather than by row position.

Code A.48. Separate features from the target, split once, and freeze it

```
from sklearn.model_selection import train_test_split

assert model_df.index.name == "customer_id"
assert model_df.index.is_unique

FEATURES = ["recency_days", "frequency", "monetary", "aov",
            "tenure_days", "store_share", "app_user", "email_opt_in"]

for forbidden in ["churned", "outcome_orders", "spend_next6m"]:
    assert forbidden not in FEATURES, f"{forbidden} was handed to the model"

train_df, test_df = train_test_split(
    model_df, test_size=0.20, random_state=42,
    stratify=model_df["churned"])       # omit stratify for a numeric target

train_ids, test_ids = train_df.index, test_df.index
assert train_ids.intersection(test_ids).empty
assert len(train_ids) + len(test_ids) == len(model_df)
```

Expected result: two frames whose row counts add to the original, and five assertions that pass silently.

Verify: the two sets share no identifier, and no outcome column appears among the features. Freeze the identities in train_ids and test_ids and reuse them; a second call to train_test_split with the same seed reproduces positions, not customers, so a re-split of a differently ordered frame grades two models on two populations.

The seed is 42 throughout the guide, and the split is 80/20. stratify keeps the outcome's base rate the same in both halves and is used for classification targets; for a numeric target, omit it.

Code A.49. Build a pipeline, fit it, and predict

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, root_mean_squared_error

model = make_pipeline(StandardScaler(), LinearRegression())
model.fit(train_df[FEATURES], train_df["spend_next6m"])
pred = model.predict(test_df[FEATURES])

print("MAE :", round(mean_absolute_error(test_df["spend_next6m"], pred), 2))
print("RMSE:", round(root_mean_squared_error(
    test_df["spend_next6m"], pred), 2))
```

Expected result: two error figures in the target's own units — dollars, here.

Verify: the prediction array is the same length as the test frame. Any step that learns something from data — a scaler, an imputer, an encoder — belongs inside the pipeline, so that it is refitted within each training fold and never learns a quantity from a customer it is about to be graded on.

Code A.50. Probabilities, and a threshold you can defend

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

# The threshold is an economic decision, derived in Chapter 9 from
# the cost of each kind of error. Never leave it at 0.5 by default.
THRESHOLD = [THRESHOLD FROM CHAPTER 9]

clf = make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000))
clf.fit(train_df[FEATURES], train_df["churned"])

p = clf.predict_proba(test_df[FEATURES])[:, 1]
treat = (p >= THRESHOLD).astype(int)

assert ((p >= 0) & (p <= 1)).all(), "a probability left the unit interval"
print("threshold:", THRESHOLD, "| treated share:", round(treat.mean(), 3))
print("AUC:", round(roc_auc_score(test_df["churned"], p), 3))
```

Expected result: the threshold you supplied, the share of the file it would treat, and the area under the ROC curve.

Verify: predict_proba returns one column per class; [:, 1] takes the probability of the positive class, and taking the wrong column produces a model that appears to be exactly backwards. Never default the threshold to 0.5 without an argument; Chapter 9 derives it from the cost of each kind of error.

Code A.51. Cross-validate every candidate on the same folds

```
from sklearn.model_selection import StratifiedKFold, cross_validate
from sklearn.dummy import DummyClassifier

CV = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

def auc_scorer(estimator, X, y):
    return roc_auc_score(y, estimator.predict_proba(X)[:, 1])

CANDIDATES = {
    "constant floor": (DummyClassifier(strategy="prior"), FEATURES),
    "recency only":   (clf, ["recency_days"]),
    "full logistic":  (clf, FEATURES),
}

rows = []
for name, (est, cols) in CANDIDATES.items():
    s = cross_validate(est, train_df[cols], train_df["churned"],
                       cv=CV, scoring={"auc": auc_scorer})
    rows.append({"candidate": name,
                 "cv_auc": s["test_auc"].mean(),
                 "cv_sd": s["test_auc"].std()})

board = pd.DataFrame(rows).set_index("candidate").round(3)
print(board)
```

Expected result: a leaderboard, one row per candidate, with a mean score and its spread across folds.

Verify: the constant floor scores about 0.5. If it does not, the scorer or the label is wrong, and every number above it is meaningless. Build the fold generator once and pass the same CV object to every candidate; candidates graded on different folds are not comparable.

Code A.52. An interpretable model with robust standard errors

```
import statsmodels.formula.api as smf

fit = smf.ols(
    "monetary ~ recency_days + frequency + tenure_days"
    " + app_user + email_opt_in + store_share",
    data=train_df).fit(cov_type="HC3")   # see Chapter 7 on robust errors

table = pd.DataFrame({"estimate": fit.params.round(2),
                      "ci_low": fit.conf_int()[0].round(2),
                      "ci_high": fit.conf_int()[1].round(2),
                      "p_value": fit.pvalues.round(4)})

print(table)
print("R-squared:", round(fit.rsquared, 3))
assert (table["ci_low"] <= table["estimate"]).all()
assert (table["estimate"] <= table["ci_high"]).all()
```

Expected result: a coefficient table with an estimate, a confidence interval, and a p-value for each term, plus R-squared.

Verify: every estimate falls inside its own interval — a cheap check that the table was assembled correctly. Use smf.logit in place of smf.ols for a binary target. cov_type="HC3" requests heteroskedasticity-robust standard errors; when that is the right request, and what it changes about the interval, is Chapter 7's.

Code A.53. A results table that joins identifiers, actuals, and predictions

```
scored = (test_df[["churned", "spend_next6m", "recency_days"]]
          .assign(p=p, treated=treat)
          .rename_axis("customer_id")
          .reset_index()
          .sort_values(["p", "customer_id"], ascending=[False, True]))

assert "customer_id" in scored.columns
assert len(scored) == len(test_df)
assert scored["customer_id"].isin(model_df.index).all(), (
    "the index was row positions, not customer identifiers")
print(scored.head().round(3))
```

Expected result: one row per scored customer, highest score first, carrying the identifier, the actual outcome, and the prediction.

Verify: the identifiers are real customers, not row numbers. rename_axis labels whatever index the frame happens to carry, so a frame read without index_col="customer_id" will pass a check for the column name and fail the check above. A ranked list without real identifiers cannot be handed to a campaign, and a score without its actual outcome beside it cannot be evaluated. The second sort key makes the ranking reproducible when scores tie.

Code A.54. A time-aware holdout: never split an ordered series at random

```
series = (marketing_daily.set_index("date").sort_index()
          .asfreq("D")["revenue"])

# asfreq inserts a row for every absent calendar day, as a missing
# value. Read that gap before deciding what it means.
assert series.notna().all(), (
    f"{int(series.isna().sum())} calendar days have no source row")

HOLDOUT_START = pd.Timestamp("[HOLDOUT START]")
train = series.loc[:HOLDOUT_START - pd.Timedelta(days=1)]
holdout = series.loc[HOLDOUT_START:]

assert train.index.max() < holdout.index.min(), (
    "the training and holdout windows overlap in time")
assert series.index.is_monotonic_increasing
assert series.index.is_unique

seasonal_naive = np.tile(train.iloc[-7:].to_numpy(),
                         int(np.ceil(len(holdout) / 7)))[:len(holdout)]
mae = np.abs(holdout.to_numpy() - seasonal_naive).mean()
print("train days:", len(train), "| holdout days:", len(holdout),
      "| seasonal-naive MAE:", round(float(mae), 2))
```

Expected result: the two window lengths and the baseline's mean absolute error.

Verify: every training date precedes every holdout date, and no calendar day is missing. A random split of a time series lets the model see next week while predicting this one, and the resulting accuracy is fiction. asfreq("D") makes an absent day visible as a gap rather than silently closing it — and a single gap left unhandled turns every error figure computed from the series into nan.

A forecast is graded against a baseline, not against zero. Which baseline, over which horizon, and by what margin a candidate must beat it are Chapter 10's decisions; the tiled repeat above is only the arithmetic that produces one of them.

A.13 Verification Patterns

This is the section to reach for after any transformation, and the one that turns predict-then-verify from a principle into a habit. Each pattern is labeled with the failure it is designed to catch, because a check that catches nothing in particular is decoration.

Two functions do most of the work. assert stops the notebook when a condition you named is false, with a message you wrote; and np.isclose compares floating-point numbers, which are almost never exactly equal even when the arithmetic is right. Use np.isclose for money and np.allclose for arrays of it.

Code A.55. Row counts and key uniqueness

```
rows_before = len(transactions)
working = transactions[transactions["quantity"] > 0].copy()

assert transactions["order_line_id"].is_unique, "order_line_id is not a key"
assert transactions["customer_id"].notna().all(), "a line has no customer"
print("rows:", rows_before, "-> after the filter:", len(working))
```

Expected result: the row count before and after, and silence from both assertions.

Verify: catches a duplicated fact table and orphaned rows. Run the uniqueness check on every fact table before computing anything, and print the count on both sides of every filter.

Code A.56. Totals before and after a transformation

```
total_before = transactions["line_revenue"].sum()
orders = (transactions.groupby("order_id", as_index=False)
          ["line_revenue"].sum())
total_after = orders["line_revenue"].sum()

assert np.isclose(total_before, total_after), "the roll-up lost revenue"
print(f"{total_before:,.2f} -> {total_after:,.2f}")
```

Expected result: the same figure twice, and silence from the assertion.

Verify: catches revenue lost to a dropped group key or multiplied by a bad join. This is the single most valuable check in the appendix, and it costs two lines.

Code A.57. Allowed categories, date ranges, and bounded shares

```
ALLOWED = {"Online", "App", "Store"}
unexpected = set(transactions["channel"].dropna()) - ALLOWED
assert not unexpected, f"unexpected channel labels: {sorted(unexpected)}"

WINDOW = (pd.Timestamp("[WINDOW START]"),
          pd.Timestamp("[WINDOW END]"))
assert transactions["order_date"].between(*WINDOW).all(), (
    "an order falls outside the declared window")

tol = 1e-9
assert transactions["discount_pct"].between(-tol, 1 + tol).all()
```

Expected result: silence. Any failure names the offending values in its message.

Verify: catches a category variant introduced upstream, a row from outside the window you declared, and a share that is not a share. The tolerance is there because a value summed from rounded components can land a fraction above 1 without being wrong.

Code A.58. Probabilities, split integrity, matrix arithmetic, and shares

```
y = test_df["churned"].to_numpy()      # the actuals, aligned to p

assert ((p >= 0) & (p <= 1)).all(), "a probability left [0, 1]"
assert train_df.index.intersection(test_df.index).empty, (
    "a customer appears in both train and test")

tp = int(((treat == 1) & (y == 1)).sum())
fp = int(((treat == 1) & (y == 0)).sum())
fn = int(((treat == 0) & (y == 1)).sum())
tn = int(((treat == 0) & (y == 0)).sum())
assert tp + fp + fn + tn == len(y), "the confusion matrix must sum to n"

shares = scored.groupby("treated").size() / len(scored)
assert np.isclose(shares.sum(), 1.0)
print(f"TP {tp}  FP {fp}  FN {fn}  TN {tn}  n {len(y)}")
```

Expected result: the four cell counts and the total, with every assertion silent.

Verify: catches a score column that is not a probability, a leaked test customer, a confusion matrix built from mismatched arrays, and a set of shares that do not partition the file.

Code A.59. Compare the code against a hand calculation

```
# By hand from the practice file, order O10003:
#   88.00 + 84.00 + 76.00 = 248.00
BY_HAND = 248.00

computed = transactions.loc[transactions["order_id"] == "O10003",
                            "line_revenue"].sum()

assert np.isclose(computed, BY_HAND), f"{computed} != {BY_HAND}"
print(f"computed {computed:,.2f}   hand {BY_HAND:,.2f}   agree")
```

Expected result: the two figures side by side, agreeing.

Verify: catches an error in the logic itself, which no internal consistency check can find. Do this once on a miniature you can compute with a calculator before running the same code on eighty thousand rows.

Table A.10 collects every check in this section and names the failure each is designed to catch, so that a check can be chosen by the risk you are worried about rather than by the command it uses.

Table A.10

Verification patterns and the failure each is designed to catch

Check	Failure it catches	Entry
Row count before and after	A filter or join that silently changed the population	A.55
Key uniqueness	A duplicated fact table inflating every total	A.55
Total before and after	Revenue lost in a roll-up or multiplied by a join	A.56
Row count unchanged by a merge	A many-to-many join expanding the table	A.33
Unmatched-key report	Dimension rows that do not exist, becoming missing values	A.34
Allowed-category set	A label variant introduced upstream	A.57
Declared date window	Rows from outside the period the claim covers	A.57
Bounded shares	A ratio computed on the wrong denominator	A.57
Probabilities within [0, 1]	The wrong predict_proba column, or a score mistaken for a probability	A.58
Train and test identifiers disjoint	A leaked customer flattering the test score	A.58
Confusion matrix sums to n	Actuals and predictions misaligned	A.58
Shares sum to 1	A partition that does not partition	A.58
Agreement with a hand calculation	An error in the logic that every internal check would pass	A.58

The evidence these checks produce is not only for you. Appendix D's verification table asks for the claim checked, the prediction made before running, the method, the expected value, the actual result, and the action taken — which is exactly what a printed row count, a reconciled

total, or a passing assertion supplies. Predict the number first, then run the check; a verification recorded after the fact verifies nothing.

A.14 Common Errors and Troubleshooting

Organized by symptom, because that is what you have when something goes wrong. Read the last line of the traceback first, find the symptom below, and work through the four steps: what it usually means, what to inspect first, a safe next step, and what not to do.

Code A.60. Read the last line first

```
try:
    result = transactions["quantity"] + transactions["channel"]
except TypeError as exc:
    print("last line of the traceback:")
    print(f"  TypeError: {exc}")
    print("types involved:",
          transactions["quantity"].dtype, transactions["channel"].dtype)
```

Expected result: the error type, its specific complaint, and the two column types responsible.

Verify: the error type names the family of the problem and the message names the instance. Printing the dtypes of the columns involved resolves most TypeError cases in one line.

A.14.1 FileNotFoundError, or No such file or directory

Usually means the file is not where the notebook is looking, or the name is spelled differently — file names are case-sensitive, and a trailing space in a name is invisible. Inspect first with Code A.5, which lists what the runtime can actually see. Safe next step: reload the file from the companion location using Code A.2, or re-upload it with Code A.3 if the runtime restarted. Do not retype the path from memory into six cells; set DATA once and build every path from it.

A.14.2 KeyError, or a column that is not found

Usually means a spelling difference, a case difference, or a column that an earlier cell renamed or dropped. Inspect first with list(df.columns). Safe next step: correct the name, or rerun the cell that creates the column, then re-run from the top so the notebook state matches the notebook text. Do not create a new column with the name you expected in order to make the error disappear; that produces a table with two versions of the same field.

A.14.3 TypeError or ValueError during arithmetic

Usually means a column that should be numeric is stored as text, typically because it arrived with a currency symbol or a thousands separator. Inspect first with df.dtypes and df["[COLUMN]"].head(). Safe next step: coerce with Code A.26 and compare the missing-value count before and after. Do not wrap the operation in a conversion inside every cell; fix the column once, near the top.

A.14.4 ValueError: Length of values does not match length of index

Usually means you are assigning an array built from a filtered or reordered frame back onto the full one. Inspect first by printing the length of the frame and of the value you are assigning. Safe next step: assign onto the same frame the values were computed from, or align on the index with .reindex before assigning. Do not use .to_numpy() or .values to silence the complaint — that discards the index alignment that was protecting you.

A.14.5 ModuleNotFoundError

Usually means the library is not present in the runtime, which in Colab is unusual for pandas, NumPy, Matplotlib, scikit-learn, and statsmodels and common for anything else. Inspect first by re-running the import cell alone. Safe next step: install it in a cell with !pip install [PACKAGE] and re-run the imports; record the install in the notebook so the next reader sees it. Do not install a package the assignment did not ask for in order to avoid writing a few lines of pandas.

A.14.6 NameError, after a runtime restart

Usually means the variable was defined in a cell that has not been run in this session. The notebook text and the runtime memory have drifted apart. Inspect first by checking whether the Connect indicator shows a fresh runtime. Safe next step: Runtime → Restart and run all, which is the only state you should ever submit. Do not hunt for the one cell to re-run; that leaves the drift in place for the next reader to discover.

A.14.7 A merge that produced more rows than it started with

Usually means duplicated keys in the right-hand table, so each left row matched several right rows. Inspect first with right["[KEY]"].duplicated().sum() and with the row count on both sides. Safe next step: resolve the duplicates in the right-hand table, then re-run the merge with validate="many_to_one" as in Code A.33. Do not drop the extra rows afterward with drop_duplicates; that keeps an arbitrary one of several matches and hides the real problem.

A.14.8 A blank or nearly empty chart

Usually means the frame handed to the plotting call is empty, or the column being plotted is entirely missing values, or a filter earlier in the cell removed everything. Inspect first by printing len(frame) and frame["[COLUMN]"].notna().sum() immediately before the plot. Safe next step: fix the upstream filter and print the excluded count, as Code A.41 does. Do not adjust the axis limits to make something appear.

A.14.9 Dates that sort or group strangely

Usually means the column is still text. ISO-formatted text such as 2025-12-31 may appear to sort chronologically, but that is a property of the format — largest unit first — rather than evidence that the column is a date; 12/31/2025 and 31-Dec-2025 sort wrongly outright, and no text column supports date arithmetic, resampling, or the .dt accessor. Inspect first with df.dtypes: a date

column reports a datetime type, not a text one. Safe next step: parse at read time with parse_dates, or convert with Code A.27 and confirm that no value failed to parse. Do not infer from one well-behaved sort that the column is a date.

A.14.10 A model that refuses text or missing values

Usually means a categorical column reached the estimator without being encoded, or a feature contains missing values that scikit-learn will not silently fill. Inspect first with train_df[FEATURES].dtypes and train_df[FEATURES].isna().sum(). Safe next step: encode the categorical explicitly as indicator columns and handle the missing values as a documented decision, inside the pipeline. Do not call .dropna() on the feature frame without counting and reporting what it removed; a model fitted on the rows that happened to be complete is a model fitted on a different population.

A.14.11 Model performance that looks too good

Usually means leakage: a feature that encodes the outcome, a transformation fitted before the split, or a customer appearing on both sides of it. This is the most consequential entry in this section, because nothing about it looks like an error. Inspect first by listing every feature and asking, for each, whether its value could have been known at the moment of prediction; then re-run the checks in Code A.48 and Code A.58. Safe next step: remove the suspect feature and refit — a real signal survives, a leak collapses. Do not report the number. Chapter 8 treats an unexpectedly excellent result as a red flag rather than an achievement, and so should you.

A.15 Saving, Exporting, and Submitting Work

The last thing that goes wrong in an assignment is usually not the analysis. It is a notebook that will not run for anyone else, a figure exported at a size no one can read, or a file that never made it off the temporary machine.

Code A.61. Export a table and a figure

```
summary = (transactions.groupby("channel", as_index=False)
           .agg(orders=("order_id", "nunique"),
                revenue=("line_revenue", "sum")))
summary.to_csv("channel_summary.csv", index=False)

fig, ax = plt.subplots(figsize=(7, 4))
ax.bar(summary["channel"], summary["revenue"])
ax.set_xlabel("Channel")
ax.set_ylabel("Revenue ($)")
ax.set_title("Revenue by channel")
fig.tight_layout()
fig.savefig("revenue_by_channel.png", dpi=200, bbox_inches="tight")
```

Expected result: two files in the working directory, and no output in the notebook.

Verify: read the CSV back and confirm its row count and total; open the PNG and confirm the axis labels are legible at the size it will be used. dpi=200 is the guide's minimum for a figure that will be placed in a document, and bbox_inches="tight" keeps the labels from being cut off.

Code A.62. Confirm the notebook reproduces from a clean runtime

```
EXPECTED_TOTAL = 1312.00      # replace with your own certified total

required = ["order_id", "customer_id", "order_date", "line_revenue"]
missing = [c for c in required if c not in transactions.columns]
assert not missing, f"columns absent from the file: {missing}"

assert np.isclose(transactions["line_revenue"].sum(), EXPECTED_TOTAL), \
    "the certified total no longer reproduces"

print("STATUS: reproduces from a clean runtime.")
```

Expected result: one status line, after Runtime → Restart and run all completes with no error.

Verify: the check has to be run after a restart, not after an ordinary execution. A notebook that only works in the order you happened to run the cells is not a notebook anyone else can use, including you next week.

A.15.1 The Submission Checklist

1. Runtime → Restart and run all completes with no error, top to bottom.
2. The environment record from Code A.12 is near the top of the notebook.
3. Every transformation cell prints or asserts its verification, and the verification log is complete.
4. Every AI exchange is documented per Appendix D, including at least one output you corrected and the evidence you used.
5. Outputs are retained or cleared according to the assignment instructions; when in doubt, retain them.
6. No credentials, API keys, personal identifiers, or confidential data appear anywhere in the notebook, including in a commented-out cell.
7. Figures and tables the assignment asks for are exported as files, at a readable size.
8. The file is named so a reader knows whose it is and which assignment it answers.
9. You have downloaded the notebook and any exported files off the runtime, and submitted the files themselves rather than a screenshot of them.

Table A.11 sets out which export format is right for which purpose. The notebook itself is the submission; the other formats are supporting material.

Table A.11

Export formats and when each is the right one

Format	Use it for	Note
.ipynb	The submitted notebook	Preserves code, output, and text together. The default.
.py	A script the course explicitly asks for	Loses all output and all text cells.
PDF	A read-only record when the assignment requires one	A picture of the work, never a substitute for it.
.csv	A table another tool will read	Always with index=False.
.png	A figure for a document or a slide	dpi=200 or higher, bbox_inches="tight".

A.16 Command Index by Task

This index is arranged by what you are trying to do rather than by what the command is called, so that it is usable before you know the function's name. The entry column points to the code entry where the pattern appears in context, with its expected result and its verification.

Table A.12 is that index. Where a task has more than one entry, both are listed, and the earlier one is usually the simpler.

Table A.12

Command index, by task

To do this	Use	Entry
Check where the runtime is and what it can see	`os.getcwd(), Path().glob()`	A.1, A.5
Load a course CSV with dates parsed	`pd.read_csv(..., parse_dates=[...])`	A.2
Upload a file from your computer	`files.upload()`	A.3
Use a file in Google Drive	`drive.mount()`	A.4
Write a table out	`.to_csv(..., index=False)`	A.6, A.61
See how many rows and columns arrived	`.shape`	A.13
See the first rows	`.head(), .tail()`	A.13

To do this	Use	Entry
Check that each column is the right type	`.dtypes, .info()`	A.14
Summarize the numeric columns	`.describe()`	A.14
Count missing values	`.isna().sum()`	A.15, A.23
List the labels in a category	`.value_counts(dropna=False)`	A.15
Confirm a column is a key	`.is_unique, .nunique()`	A.16
Keep some columns	`df[["a", "b"]]`	A.17
Keep some rows	`df[condition], .isin()`	A.18
Sort reproducibly	`.sort_values([measure, id])`	A.19
Build a derived column	`df["new"] = ...`	A.20
Rename a column	`.rename(columns={...})`	A.21
Recode categories	`.map(dictionary)`	A.21
Work on a subset safely	`.copy()`	A.22
Fill or drop missing values	`.fillna(), .dropna(subset=[...])`	A.24
Find duplicates	`.duplicated(subset=[...])`	A.25
Turn currency text into numbers	`pd.to_numeric(..., errors="coerce")`	A.26
Turn text into dates	`pd.to_datetime(..., errors="coerce")`	A.27
Pull year, month, or weekday out of a date	`.dt.year, .dt.to_period("M"), .dt.day_name()`	A.27
Summarize by group	`.groupby(...).agg(name=(col, func))`	A.29
Count distinct orders or customers	`("order_id", "nunique")`	A.29, A.31
Compute shares that sum to 1	`col / col.sum()`	A.31
Roll lines up to orders	`.groupby("order_id").agg(...)`	A.32
Join a dimension onto a fact table	`.merge(..., validate=, indicator=)`	A.33

To do this	Use	Entry
Find keys that did not match	`set(left) - set(right)`	A.34
Go wide	`pd.pivot_table()`	A.36
Cross-tabulate two categories	`pd.crosstab(..., normalize=)`	A.37
Go long	`.melt()`	A.38
Build a customer-grain table	`two-step groupby`	A.39
Draw a distribution	`ax.hist(), ax.boxplot()`	A.41, A.45
Compare groups	`ax.bar()`	A.42
Show a trend	`ax.plot()`	A.43
Show a relationship	`ax.scatter()`	A.44
Add a baseline to a chart	`ax.axhline(), ax.axvline()`	A.46
Repeat a chart per group	`plt.subplots(1, n, sharey=True)`	A.47
Split into train and test	`train_test_split(..., stratify=)`	A.48
Scale and fit in one object	`make_pipeline()`	A.49
Predict a number	`.predict()`	A.49
Predict a probability	`.predict_proba()[:, 1]`	A.50
Compare candidates on identical folds	`cross_validate(..., cv=CV)`	A.51
Fit a readable regression	`smf.ols(...).fit(cov_type="HC3")`	A.52
Build a scored list for a campaign	`.assign(p=...).reset_index()`	A.53
Hold out the end of a time series	`.loc[:cut], .loc[cut:]`	A.54
Stop the notebook when a check fails	`assert condition, "message"`	A.55
Compare two money totals	`np.isclose(), np.allclose()`	A.56
Save a figure	`fig.savefig(..., dpi=200)`	A.60

References

The library documentation is authoritative for every argument named in this appendix, and the works below are the citable references for the packages introduced in Table A.4 and for the

Colab environment described in Section A.1. This appendix records the recurring subset the guide uses; it does not replace the reference manuals.

Google. (n.d.). *Google Colaboratory: Frequently asked questions*. Retrieved August 2, 2026, from Google Colab frequently asked questions

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature, 585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering, 9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

McKinney, W. (2010). Data structures for statistical computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.

Seabold, S., & Perktold, J. (2010). Statsmodels: Econometric and statistical modeling with Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 92–96).