3H EDUCATION ACADEMY

171
COURSE TEXTBOOK

REPRODUCIBLE PROJECT GUIDES

Writing a Reproducible Project README



SETUP / STEPS

DIGITAL & TECHNOLOGY

3H-IT-0171

STUDENT EDITION

Writing a Reproducible Project README

Learning outcomes

- Document a supplied local project so another learner can reproduce it.
- State inputs, commands, expected outputs and limitations precisely.
- Validate the README from a clean copy instead of relying on author memory.
- Audit README claims against source behavior and fixture arithmetic.
- Distinguish documented input expectations from checks actually enforced.
- Design safe repeat runs and diagnostic negative cases without deleting outputs.
- Record clean-copy evidence with expected, observed and not-run states.
- Deliver a complete first-reader README with bounded compatibility claims.

Original training material with synthetic examples. Local practice only; no certification or production-readiness claim.

Study the foundations, work through the applied units, complete the practice tasks, and then attempt the assessment. Use only the supplied fictional practice data. Keep your original inputs and your revised work separate.

Core concepts

Write for a reader who does not know the project

A README is an entry point to a reproducible task. Begin with what the program does and what it does not do. The supplied project totals integer quantities in a synthetic CSV and writes a short text result; it does not edit the input, connect to a service or validate arbitrary business data. Name the required runtime and explain that the program uses only the Python standard library. Give the folder structure and identify which file is input, which is source code and which is generated output. Avoid instructions such as “run it as usual,” because the reader has no access to your habits.

Make commands and expected results checkable

State the working directory before giving a command. Explain the two positional arguments in the order the program expects: input path, then output path. Show a copyable example and the exact text expected in the output. Separate console behavior from file contents; a successful silent program may still produce its required file. Because this program uses exclusive output creation, the README must explain that an existing output path causes failure and that a fresh filename should be used. Do not advise deleting unrelated files or forcing an overwrite. Document supported UTF-8 CSV headers and integer values so the reader knows the input contract.

Test the document as an interface

Copy the project to a fresh directory and follow only the README. Record the actual Python version and operating environment used. Verify the output independently from the small input: 2+5+3 is ten. Then try a missing input path and a pre-existing output path to see whether troubleshooting advice matches observed behavior. Do not include a successful run claim until a run was performed. A README can be correct for the supplied fixture while still lacking broad cross-platform testing; state that boundary. Reproducibility improves when setup assumptions and limitations are explicit rather than hidden in the author’s environment.

Worked example

Worked example: “From the folder containing total.py and items.csv, run python total.py items.csv result-new.txt. A successful run creates result-new.txt containing Total quantity: 10 followed by a newline. It may print nothing. Use a new output filename if that path already exists.” This is more useful than “Run the script and check the result,” because it identifies location, command, arguments, artifact and expected value.

Foundation practice

Files

Items.csv: id,quantity
A,2
B,5
C,3

Total.py

```
import csv
import sys
from pathlib import Path
if len(sys.argv) != 3:
    raise SystemExit("Usage: python total.py INPUT.csv OUTPUT.txt")
source, target = map(Path, sys.argv[1:])
if source.resolve() == target.resolve():
    raise SystemExit("Input and output must differ")
with source.open(newline="", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    if reader.fieldnames != ["id", "quantity"]:
        raise ValueError("Expected id,quantity header")
    total = sum(int(row["quantity"]) for row in reader)
with target.open("x", encoding="utf-8") as f:
    f.write(f"Total quantity: {total}\n")
```

Environment

Python 3.10 or newer; standard library only. Check your installed python command; on some systems use python3. No installation is required if already available.

Deliverable: README.md or README.txt plus a reproduction log from a fresh folder.

Foundation tasks

- Draft purpose, file map, runtime, input schema, run command, expected output and limitations.
- Ask a second learner to follow the README, or emulate a fresh-reader run from a copied folder.
- Capstone: deliver the README and observed run log, including the successful result and clear handling of an existing output file.

01 Define the reproducible task before writing

A project README is a small operating agreement between its author and an unfamiliar reader. The reader needs to know what will happen, what must already exist, what they must type, and what evidence will distinguish success from a misleading appearance of success. A friendly introduction is useful, but it cannot substitute for these operational facts. In this course the project is deliberately small: a Python program reads a local CSV file, converts the quantity column to integers, sums those integers, and creates a new text file. The small size lets you check every claim rather than relying on impressive but unverified descriptions.

Start with a one-sentence purpose that names both input and output. “Calculate the sum of integer quantities from a supplied UTF-8 CSV and save a local text summary” is precise. “Manage inventory” is too broad: the program does not track stock locations, authenticate users, prevent duplicate identifiers, or maintain a history of transactions. Those absent capabilities matter because a reader might otherwise treat a useful arithmetic exercise as a business system. Scope statements prevent mistaken expectations before troubleshooting becomes necessary.

Define reproducibility at the level of this task. Another learner, using the supplied files and a suitable installed Python runtime, should obtain the same calculated result for the same input. Their absolute folder path, Python patch version and error traceback formatting may differ. Your README should identify which differences are harmless and which invalidate the result. Matching the final total while silently changing the input is not a reproduction of the supplied case. Matching the file content with an honestly recorded environment is stronger evidence than copying a screenshot of someone else's terminal.

A practical writing order is to inspect the files, build a fact sheet, perform a clean run, and then write the shortest complete route through those facts. This does not mean documenting every line of code. Explain the interface the reader must use and the assumptions that can cause failure. Keep purpose, prerequisites, file roles, invocation, result, safe retry, limitations and verification distinct enough to find quickly. Readers scan when they are blocked; a paragraph that mixes all eight topics makes otherwise correct information unnecessarily difficult to use.

02 Build a source-backed interface inventory

Before drafting commands, inspect the supplied total.py rather than inferring behavior from its filename. The imports name csv, sys and pathlib, all Python standard-library modules. The script requires exactly two positional arguments after its own filename. It maps those arguments to source and target paths in that order. It rejects paths that resolve to the same location, then opens the input as UTF-8 with CSV-aware newline handling. The header comparison is exact: the two column names must be id and quantity, in that order. The program converts each quantity with int and adds the converted values.

Notice what the code does after summing. It opens the output with mode x, which means create a new file and fail if the path already exists. It writes one line containing the label Total quantity, a colon, a space, the calculated integer, and a newline. It does not call print on the successful path. A reader who expects terminal output could mistake success for inactivity, then repeat the command and encounter a file-exists failure. The README should prevent this predictable sequence by saying where to inspect the result the first time.

Create an evidence inventory in which each documentation claim has a source. “The argument order is input then output” follows the source-target assignment. “The fixture total is ten” follows independent arithmetic on items.csv. “My run used Python 3.12.3” can come only from an actual environment observation; neither the script nor this chapter establishes your installed version. “The program is safe for every CSV” has no support at all. Separating these evidence types stops plausible guesses from becoming instructions.

An inventory also exposes constraints that deserve testing. The script checks the header but does not explicitly count row fields, require unique identifiers, or reject negative quantities. Do not convert a narrow header check into a claim of complete validation. Documentation may recommend a stricter supported input shape than the program happens to accept, provided it clearly distinguishes the recommended contract from enforced checks. This distinction is especially important when a tool is used for teaching rather than deployed as a general data-processing service.

Claim	Evidence source	README consequence
Input precedes output	Argument assignment in total.py	Show both arguments in order
Output must be new	Output open mode x	Give a non-destructive retry
Fixture total is 10	2 + 5 + 3	Show exact expected file line
No success console text	No print on success path	Tell reader to open output
All IDs unique	Not checked by this script	Do not claim enforcement

E02-X1 Audit six README claims

Read assets/bad-readme.txt and the supplied program. Identify the six operational defects: runtime precision, argument order, output channel, overwrite behavior, validation scope and compatibility evidence. Replace each claim with a concrete supported sentence.

Submit: Six numbered corrections, each linked to a code fact or required observation.

03 Design a file map that survives a fresh copy

A file map is a compact explanation of what the reader received and what the program will generate. For the foundation fixture, total.py is source code, items.csv is input data, and result-new.txt is a future output, not a required input. README.md or README.txt contains instructions. This distinction seems obvious when all files are open in the author's editor, but it disappears when someone receives a folder containing old reports, temporary files and several similarly named scripts. The README must not depend on the arrangement of your desktop.

Use relative paths from a stated project root. Here, project root means the folder containing total.py and the input CSV chosen for the run. It does not mean the computer's home folder or the root of a drive. If a reader copies the project elsewhere, relative paths still work after they open a terminal in the copied folder. By contrast, an example pointing to a personal absolute path may fail immediately or accidentally read a different input. Explain any subfolder that actually exists; do not invent a data directory merely because other projects use one.

Generated outputs should be identified as disposable only within this synthetic exercise and only when the learner knows which files they created. A safer README does not require deletion at all. It gives a new output filename for a second run, such as result-second.txt. This preserves the first result as evidence and avoids training readers to remove files indiscriminately. A naming convention can make runs easy to compare, but the program itself does not enforce that convention.

A fresh-copy test should copy the required source and input files, not every artifact from a previously successful run. If the copied folder already contains result-new.txt, the recommended command will fail correctly because exclusive creation protects that path. That failure says more about the package or instructions than about the arithmetic. List the initial file set explicitly, then describe the generated file separately. This lets another learner verify that their starting state matches the documented one before interpreting an error.

```
project-copy/
  README.txt       instructions written by the learner
  total.py         supplied program, unchanged
  items.csv        supplied input, unchanged

Created after a successful example run:
  result-new.txt   generated text result; absent before the run
```

04 State prerequisites without hiding setup work

A prerequisite is something the reader must already have before the example can run. The exercise specifies Python 3.10 or newer and uses only the standard library. Therefore no third-party package installation belongs in the reproduction instructions. Adding a package manager command because a familiar tutorial includes one would create unnecessary work and a new source of failure. The task is to document the supplied program, not to turn it into a differently managed project.

Interpreter names vary between systems. A command named python may select the intended Python installation, a different interpreter, or nothing at all. Some environments use python3. Ask the reader to check the chosen command with its version option, then use that same command consistently. If python --version reports an unsuitable version, repeating the program command does not fix the prerequisite. Record the blocker honestly and use an already available suitable interpreter if one exists. Installing software is outside this practice task.

Distinguish a minimum stated requirement from an observed test environment. “Requires Python 3.10 or newer” is the exercise contract. “Tested with Python 3.11.8 on this recorded operating system” is an observation only if that test happened. A minimum version does not establish that every later release, filesystem, terminal and permission configuration has been tested. Keep broad compatibility claims out of the README unless supported by a real test matrix.

Other prerequisites are more mundane but often more important than version numbers. The reader needs read access to the input, write access to the directory where the new output will be created, and the supplied file names spelled correctly. The script does not create missing output-parent directories. It also does not open an editor or locate the input automatically. Mention these boundaries in plain language rather than overwhelming the reader with platform-specific permission commands. A good prerequisite section helps a reader recognize whether they can start; it does not ask them to perform unrelated system administration.

```
python --version
# If the installed suitable interpreter is named python3 instead:
python3 --version
# Use the selected command consistently in later examples.
```

05 Make command syntax unambiguous

A copyable command should be complete enough to execute without guessing and modest enough not to surprise the reader. For this program, the basic form is interpreter, script path, input path, output path. The first positional path after total.py is read; the second is created. The names INPUT.csv and OUTPUT.txt can explain the pattern, but a first-run example should use real supplied names rather than placeholders that a beginner might type literally. State the working folder immediately before the example so relative paths have a defined meaning.

Explain which parts are commands and which are comments or expected results. A terminal prompt character is not part of the command; omitting prompts from copyable examples avoids accidental pasted punctuation. A printed output line is not another shell command. Similarly, the words “or python3” should not be appended to a line meant to be copied. Show one clear route, then explain that the reader may replace the interpreter name throughout if that is how their installed runtime is invoked.

Paths with spaces require appropriate shell quoting. The simplest course route uses the supplied filenames without spaces. A documentation note can still explain that a path such as input files/lot.csv must be passed as one argument, usually by quoting it in the reader's shell. Do not describe one quoting convention as universal across every shell. The Python rehearsal script supplied later avoids shell parsing by passing arguments as a list, which is a useful way to test the program interface independently of terminal syntax.

Working-directory mistakes deserve specific guidance. If the terminal is in a different folder, the script may not be found before Python even begins reading the CSV. If the script path is correct but the input path is wrong, the program can start and then fail opening the input. These are different failures with different remedies. The README should direct the reader to check location and names, not to change the data or use elevated permissions as a first reaction. Reproducible instructions reduce ambiguity rather than offering a long menu of unrelated fixes.

```
# From the folder containing total.py and items.csv:
python total.py items.csv result-new.txt

# For a second run, use a different, not-yet-existing target:
python total.py items.csv result-second.txt
```

E05-X1 Write a runnable first-run section

Write a first-run section for assets/lot.csv. Use total.py unchanged, choose lot-first.txt as the new output and explain the meaning of both positional arguments. Calculate the result independently. State what is expected on stdout and in the output file.

Submit: A short README section with location, command, argument explanation, expected file text and safe repeat command.

06 Document the input contract honestly

An input contract tells the reader which data shape to supply. The foundation program expects UTF-8 CSV with the header id,quantity. Header order matters because the code compares the entire fieldname list with an exact two-element list. A file headed quantity,id is not accepted merely because it contains the same words. Capitalization and additional header spaces also matter. This is an easy place for a README to be more helpful than a traceback: show the required header as literal text and include a small valid sample.

The quantity cells are converted using Python's integer conversion. The ordinary supported examples use whole-number strings, including zero and signed integers. A decimal string such as 2.5 does not become two or three automatically; it produces an error. The program does not silently skip malformed quantities, and it does not calculate a partial output before

discovering them. Summation finishes before the target is opened. That ordering lets the README explain why an invalid input should not produce a new report file in these fixtures.

The id field is present for the sample records but is not used in the calculation. Duplicate identifiers are not rejected by this implementation. In assets/duplicate-id.csv, both rows have L1, and quantities four and seven are still added. This behavior is not a recommendation to use duplicate identifiers in real records. It is a limit that must not be concealed behind the phrase “validates the CSV.” A precise statement is that the script checks the header and converts quantity values, while wider business rules remain outside scope.

There is another subtle distinction between acceptance and a supported contract. A row with an extra trailing field may be parsed by DictReader with additional data under a special key that this script never inspects. The script can therefore calculate a result while ignoring that extra field. Your README can require exactly two fields per row for supported use, but should not say the script enforces that requirement. Readers need to know whether a rule is a documented expectation or a runtime safeguard. Test the narrow behavior before deciding which words belong in the manual.

Fixture	Relevant feature	Expected behavior
items.csv	2, 5, 3	Create total 10
lot.csv	4, 0, -2, 7	Create total 9
header-only.csv	Header, no records	Create total 0
wrong-header.csv	Reversed header	Fail before output
fraction.csv	Quantity 2.5	Fail before output
duplicate-id.csv	Repeated id	Create total 11; no uniqueness check
extra-field.csv	Extra trailing value	Create total 4; extra value unexamined

07 Describe outputs as observable artifacts

An expected result should name the artifact and its exact meaningful content. For items.csv, the file line is Total quantity: 10 followed by a newline. Capitalization, spacing and the label are part of the observable interface. If a README merely says “you should get ten,” a reader cannot tell whether ten should appear in the terminal, a CSV, a report window or an exit code. The program's success exit status and its numerical total are different things: a successful calculation of ten normally exits with status zero.

Text files have both logical content and byte representation. The script writes a newline through a text stream. Depending on the operating environment, newline translation can affect the physical bytes written, while reading the file as ordinary text gives the same logical line. For this course, compare the decoded text and explicitly name the trailing newline. Do not demand a particular byte sequence across operating systems unless you have made

byte-level output part of the contract and tested it. A README should be precise about the right level of equivalence.

Inspect the created output rather than assuming it exists because no error was visible. Some terminal environments hide output or retain old text from another run. A file from yesterday can also look correct while the latest command failed. Using a new filename makes provenance clearer: the expected file must be absent before the run and present afterward. The reproduction log should tie the exact command to that artifact and record the exit code where the test method exposes it.

The output is a summary, not a copy of the input. It omits identifiers, individual quantities and the source filename. Those omissions are intentional characteristics of the supplied program. If the reader needs a traceable report that includes every record, this program does not supply it. Explain that limitation rather than implying that a single total independently proves all business data were correct. An arithmetic check supports the total for the fixture; it does not establish completeness, authenticity or suitability of an external dataset.

08 Explain repeat runs without destructive cleanup

Reproducibility often requires a second run, but the first run changes the filesystem by creating an output. Instructions that work only once are incomplete unless they explain that changed state. This script deliberately uses exclusive output creation. Repeating the same command with the same target causes a file-exists failure, even if the input is unchanged and the expected arithmetic is still correct. That is a protection against replacement, not evidence that Python has stopped working.

The simplest repeat policy is to choose a fresh filename. For example, preserve result-new.txt and create result-second.txt. Both should contain the same logical text when the input is unchanged. If the results differ, investigate the input and source files before deciding which result to trust. Do not overwrite one report with another and then claim a comparison was performed. Keeping both artifacts makes the observation easier to audit and avoids unnecessary deletion instructions.

A strong troubleshooting entry states the symptom, probable cause, check and bounded remedy. “FileExistsError: the chosen output path already exists. Confirm that the named path is your earlier output. Choose a different, absent target name and rerun with the same input.” This is better than “delete the result and try again,” because the latter assumes the reader has correctly identified the file and is willing to lose its contents. Documentation should not normalize destructive recovery when an equally simple non-destructive route exists.

Do not promise stronger filesystem protections than the source establishes. The explicit same-resolved-path check prevents the obvious case of naming the input as output, and exclusive creation refuses an existing target. The course does not analyze every unusual filesystem relationship, concurrent writer or permission race. State the practical local behavior demonstrated by the fixture. A beginner README can be both useful and bounded: it need not become a security treatise, but it must not turn a narrow protection into an absolute guarantee about arbitrary environments.

E08-X1 Turn failures into safe troubleshooting

Use the hypothetical records in assets/observation-records.json. They are supplied teaching records, not your own runs. For R2, R3 and R4, write a troubleshooting row with symptom, cause, safe next step and output-state expectation. Keep the original successful result from R1 intact.

Submit: Three complete troubleshooting rows plus a sentence distinguishing supplied records from personal observation.

09 Read errors without documenting unstable details

A helpful README does not need to reproduce an entire traceback. Tracebacks can include absolute paths, line numbers, runtime-specific formatting and details that change when a file is moved. These are useful for investigation but poor foundations for universal instructions. Document stable failure categories and the code conditions that trigger them. For the supplied cases, FileNotFoundError identifies an unavailable input path, FileExistsError identifies a target that already exists, and ValueError covers the incorrect header or a quantity that cannot be converted to an integer.

Different mistakes can share an exception category. A wrong header and a fractional quantity both raise ValueError, but the appropriate remedy differs. The header error contains the program's explicit expected-header message. The conversion error arises from int on the offending value. Teach the reader to examine the immediate message and the input contract rather than treating every ValueError as the same event. A troubleshooting table should separate these cases and name a check that distinguishes them.

The program also uses SystemExit for argument count and input-output equality. If the reader omits an argument, the usage message names the two required paths. If the reader gives the same resolved path for input and output, the program reports that they must differ. Neither case should be documented as a successful run with no output. A nonzero exit status is part of the observation. When running manually, record the visible message honestly; when using the supplied rehearsal script, the return code is captured directly.

Avoid advice that changes permissions, installs packages or rewrites the source before checking the simple interface facts. A typo in a path is not a missing dependency. An already existing target is not a reason to run with administrator privileges. The smallest safe remedy is usually to correct the command or choose a new output name. If the actual error is outside the documented cases, record it as an unresolved observation rather than forcing it into a familiar category. Reproducibility includes knowing where the documented evidence stops.

10 Construct a clean-copy reproduction protocol

A clean-copy run tests the document rather than the author's memory. Create a new temporary or otherwise isolated practice folder and copy only the required source and input files into it. Read the README as if the original folder were unavailable. Follow its stated working-directory instruction, choose the documented interpreter command, and use an output filename that is absent. Do not quietly fix omitted steps and still mark the README complete. Each intervention is evidence that the instructions need a correction.

Before execution, record the starting files and the fact that the target does not exist. Record the interpreter version returned by the interpreter actually used. Then run the exact command and inspect the result. For items.csv, independently add two, five and three. Compare the observed output text with the expected line and confirm that the input content is unchanged. A successful process alone is insufficient; the wrong command could process another CSV successfully.

After the happy path, test a small set of failures that the README claims to explain. Repeat with the same target to observe refusal to replace it. Use a nonexistent input name with a fresh target. Use the supplied wrong-header and fractional-quantity fixtures separately. The tests should be isolated so one case does not accidentally inherit an output from another. This is why the rehearsal script creates separate case directories, rather than running every command against the same collection of old results.

Do not label an unexecuted case as passed. If the required interpreter is unavailable, record not run and the prerequisite blocker. If you manually inspect a code path but do not execute it, call that source inspection. If a partner follows the README independently, record the partner's observed outcome without inventing details of their environment. The value of the reproduction log is its honesty about what happened, not the number of green labels. A short accurate log is more useful than a long checklist filled with assumed successes.

11 Use the supplied rehearsal script as evidence, not a shortcut

The assets include rehearse.py, a standard-library runner that creates temporary copies and invokes total.py using the same Python interpreter that runs the rehearsal. It prints one JSON record for each case. The runner does not install dependencies, access a network, or modify the supplied fixtures. All generated reports are placed inside newly created temporary directories and removed when those temporary contexts end. The printed records are the evidence to retain. Run it from any working folder by giving the correct path to the script.

Read the runner before using it. It copies total.py and one selected fixture into a case directory, optionally creates a protected target, captures standard output and standard error, then reports the return code and observed target text. Because it uses subprocess with an argument list, it does not rely on shell quoting. Its path to the assets comes from the runner's own location. These choices reduce hidden assumptions, but they do not test every way a human may type a command into a terminal.

The runner is a diagnostic aid, not a replacement for your first-reader exercise. Your README should still contain a clear manual command and artifact expectation. Automated checks can show that the program behaves as described under specific inputs, while a manual clean-copy run checks whether the prose actually tells a reader what to do. If the script passes but a reader cannot find the correct folder, the documentation still needs improvement.

Record the distinction between expected case outcomes printed in this chapter and the records produced by your execution. A process may fail for a different reason on your machine, such as a missing copied file or permission restriction. Compare the result before calling it successful. Negative tests pass as tests when the intended rejection occurs and the protected data remains unchanged; they are not successful calculations. This distinction prevents an exit code of one from being misreported as a broken rehearsal when rejection was the intended observation.

```
python assets/rehearse.py
# If already inside assets, the equivalent command is:
python rehearse.py
```

12 Keep a reproduction log with useful provenance

A reproduction log should be detailed enough for a reviewer to reconstruct what was tested, but not full of irrelevant personal information. Record the interpreter version, operating environment, fixture name, exact command or argument list, exit status when available, standard output, relevant error category, output artifact content and whether the input changed. Use a neutral label such as fresh-copy-1 for the temporary folder; a personal home-directory path is usually unnecessary. Do not record passwords or unrelated environment variables.

Separate expected from observed columns. Expected means what the program and fixture lead you to predict. Observed means what a particular run actually produced. A result can then be classified as matched, mismatched or not run. Without that distinction, a copied expected line can masquerade as evidence. The exact same text can be useful in both columns, but only after an actual observation establishes the second one.

A log should also preserve the sequence of stateful cases. The existing-output case starts with a target already present, so record its initial content and verify that the same content remains afterward. Otherwise an error message alone does not prove the file was protected. For the missing-input and fractional-quantity cases, record that the fresh target remained absent. These observations connect the failure to its filesystem consequences, which is what a reader needs for safe troubleshooting.

Do not demand a predetermined runtime version in another learner's log. The acceptance rule is an eligible actual interpreter with the stated behavior, not an invented match to a sample environment. Likewise, generated temporary paths and traceback line numbers are not universal answers. A model log should identify fixed expected values and environment-dependent fields separately. This lets a reviewer grade correctness without pressuring learners to fabricate observations that resemble the teacher's computer.

Case	Before	Expected exit/result	Important after-check
Valid items	Target absent	0; total 10; stdout empty	Input unchanged; target present
Existing target	Target contains KEEP	Nonzero; FileExistsError	Target still KEEP
Missing input	No input; target absent	Nonzero; FileNotFoundError	Target absent
Fraction	quantity is 2.5	Nonzero; ValueError	Target absent
Same path	Input used as target	Nonzero; paths must differ	Input unchanged

E12-X1 Plan an auditable reproduction log

Using the five cases in the preceding table, prepare a complete expected-results ledger before running anything. Then, if a suitable interpreter is available, run assets/rehearse.py and add observed outcomes. If it is not available, label observations not run. Do not copy an expected runtime version into the observation column.

Submit: Five expected rows, a real or explicitly not-run observation section and a conclusion about which README claims were checked.

13 Organize the README for first use and recovery

A useful README has a reading route, not merely a collection of facts. Put a concise purpose first, followed by prerequisites, files and the first successful command. Show the expected artifact immediately after that command. Explain repeat runs and common failures after the reader understands the normal path. Finish with limitations and the scope of observed testing. This order answers the beginner's questions in the sequence they are likely to arise without burying the command below a long project history.

Headings should name the reader's task. “Run the sample” is easier to navigate than “Operational modalities.” “If the output already exists” is more useful than a generic “Miscellaneous.” A plain-text README can use short labels and line breaks; Markdown can add headings and code blocks when the file is rendered by a suitable reader. Formatting is a means of finding accurate content, not evidence that the commands work. The same operational facts should remain understandable when viewed as unformatted text.

Avoid duplicated commands that gradually diverge. If a quick-start section uses result-new.txt but troubleshooting instructs the reader to inspect report.txt, the document has created an avoidable contradiction. A single canonical first-run command and one clearly labeled repeat example are enough for this small project. When explaining placeholders, keep them distinct from real filenames. Every referenced asset should exist in the delivered folder or be explicitly described as a generated output.

Accessibility here includes readable terminology and unambiguous alternatives. Explain standard output as the program's normal terminal text, and explain a positional argument as a value whose order determines its role. Do not rely solely on color to distinguish commands

from results. Do not place essential warnings only in an image. A reader copying from plain text should still know which file is input, which is output, and which target must be absent. Clear language and correct structure reinforce each other.

14 Review a README as a set of testable claims

Review documentation by asking whether each important sentence can be traced to the source, a fixture calculation or an actual run. This method is more reliable than judging tone or length. A polished paragraph that says the program checks all identifiers is wrong, while a plain sentence that accurately limits validation is useful. Mark claims as supported, contradicted or not established. Then repair the claim itself rather than adding a vague disclaimer at the end.

Consider the statement “The command prints the total and overwrites old reports.” It contains two independently false claims for this program. Changing it to “The command usually produces a result” hides the errors instead of fixing them. A strong correction names the output file and exclusive-creation rule. Similarly, “Any CSV works” should become a specific header and quantity contract with explicit non-goals. Precision is not verbosity when it removes an actual ambiguity that would block reproduction.

Cross-check the file map against every command and troubleshooting row. Verify that the command's output name matches the expected-artifact section. Check that the stated argument order matches the script. Check that example arithmetic uses the listed input values, not a previous example. These cross-references catch documentation drift even when every paragraph seems reasonable in isolation. A review table with claim, evidence and correction is a practical way to keep the reasoning visible.

Finally, distinguish a document defect from a program limitation. The lack of duplicate-ID validation is a program limitation to describe, not a README bug to conceal by changing code during the documentation exercise. A reversed command in the README is a documentation defect to correct. If a requested feature would require source changes, state that it is outside this project's current behavior. Keeping those categories separate prevents a reproduction task from silently becoming a software modification task.

E14-X1 Decide what the evidence supports

For each assertion, label supported, contradicted or not established and give the exact reason: (1) items.csv produces total 10; (2) duplicate identifiers are rejected; (3) a header-only input produces zero; (4) a successful run proves every operating system works; (5) a fresh target avoids the known existing-output failure; (6) an extra trailing field is always rejected. Use the supplied code and fixtures, not assumptions about other CSV tools.

Submit: Six decisions with concrete supporting facts.

15 Maintain instructions when a fixture changes

A README can become stale even when the program remains unchanged. Suppose a new practice input has quantities four, zero, negative two and seven. The command structure stays the same, but the fixture name and expected total change. Copying the old expected ten into the new example would make a correct run appear wrong. Treat each example as a small linked unit: input name, input values, command, target name and expected content must agree.

Some changes affect the interface rather than only the expected result. If a future program changes argument order, output format or overwrite policy, the related README sections and tests must change together. Do not assume the current README describes a different program simply because the filename is still total.py. Compare the current source and rerun the relevant cases. This course uses the supplied unchanged program, so any discussion of future behavior must remain hypothetical rather than being written as an existing feature.

A useful maintenance checklist asks which claims depend on a changed fact. A new fixture affects arithmetic and example names. A new dependency affects prerequisites and setup. A new validation rule affects accepted inputs and troubleshooting. A changed output channel affects the success instructions. This dependency view is more efficient than rereading every sentence without a purpose, but it still requires a final consistency review to catch indirect references.

Keep observed test evidence tied to the actual version of the files used. A log from an earlier input is not proof that a revised example works. You can preserve the old log as history, but label it clearly and add a new run for the current case. If no new run has occurred, state that the revised expectation was calculated or inspected but not executed. Documentation maintenance should improve the reader's confidence by preserving provenance, not by recycling a successful-looking record detached from its inputs.

16 Work through a documentation failure investigation

Consider a fictional learner, Mina, who receives only total.py, items.csv and a short note saying “run Python and look for ten.” Mina opens a terminal in a different folder and types python total.py. The interpreter cannot find the script. That result does not test the CSV or the arithmetic. The first documentation defect is missing location information, and the second is missing argument information. A reviewer should repair those omissions before diagnosing the program itself. Otherwise an error that occurs before the script starts may be confused with a defect in the script's calculations.

Mina then opens the correct folder and runs python total.py items.csv report.txt. The command completes without terminal text. She assumes nothing happened and repeats it. The second command reports FileExistsError. The report from the first run contains the correct total. The documentation failed to identify the output channel and repeat-run state, while the program behaved according to its design. A complete incident account therefore says that the first calculation succeeded, the second invocation was correctly refused, and

the reader lacked instructions for inspecting the first artifact. Calling the whole episode “the program failed twice” would misrepresent the evidence.

Now imagine that a colleague sends Mina an old report.txt with the expected total already inside. If she copies that report into her clean folder before following the corrected instructions, the first command will fail because the target already exists. Seeing ten in the old file does not establish that her command succeeded. The necessary evidence includes the target's absence before execution, the actual return status, and the content after execution. This is a provenance problem rather than a numerical problem: the right value can still belong to the wrong event.

The corrected README should address the causal chain rather than merely adding a generic warning. It should name the starting folder, list the initial files, show both arguments, state that stdout is empty on success, name the newly created report, and explain fresh filenames for repeats. The reproduction log should preserve the sequence of observations instead of collapsing them into a single pass/fail label. This case illustrates why troubleshooting quality depends on understanding when a failure occurs, which state already exists, and which artifact belongs to which command.

Use the same method when reviewing a real peer's report. Ask for the smallest missing fact: which folder, which command, which target state, or which observed message? Do not ask them to rerun unrelated commands or change system settings without a reason. Once the missing fact is supplied, revise the instructions that would have prevented the confusion. A README is effective when it reduces avoidable inference, not when it simply contains more warnings.

17 Compare text precisely without overclaiming byte identity

A reader can compare the expected output visually, but a small Python check makes invisible differences easier to explain. The expression repr(text) displays a representation in which a trailing newline is visible as a backslash-n sequence. That display is not the literal content a user should type into the output file. For the sample report, ordinary text reading yields the string represented as 'Total quantity: 10\n'. If a learner writes the two characters backslash and n instead of an actual newline, the file is not the same logical text.

Use read_text with an explicit encoding when inspecting the generated result. The program writes UTF-8, so the inspection should decode UTF-8 rather than relying on a machine's implicit default. Comparing the decoded string with an expected string is an appropriate local acceptance check for this exercise. It avoids treating platform newline translation as a different arithmetic result. A byte-level comparison answers a narrower question about exact encoding and line endings; it can be useful, but the README should not silently substitute that question for the documented logical text contract.

The following small inspection script is intended to be saved as inspect_result.py in the copied project folder after running the example. It reads only the generated report and prints its representation and a Boolean comparison. It does not recreate the report or repair it. If the report is absent, the read fails, which is useful evidence that the preceding command did

not produce the expected artifact. If the Boolean is False, inspect the displayed representation and the actual command before changing anything. A comparison tool should reveal a discrepancy, not hide it by rewriting the expected file.

```
from pathlib import Path

actual = Path("result-new.txt").read_text(encoding="utf-8")
expected = "Total quantity: 10\n"
print(repr(actual))
print(actual == expected)

# Expected display after the valid sample run:
# 'Total quantity: 10\n'
# True
```

This check is deliberately separate from independent arithmetic. It verifies that the artifact matches the expected string, but the expected string could itself be wrong. The arithmetic 2 + 5 + 3 = 10 supplies a second reason to trust the expectation. Conversely, an arithmetic total of ten does not prove that the output label, spacing or newline is correct. Combining independent arithmetic with artifact inspection checks both the content calculation and the documented output interface.

Do not make the inspection script a hidden prerequisite for basic use. A reader can open the text file in an editor and compare the line. The script is an optional precise check for the learning exercise and for recording reproducible evidence. If you include it in a handoff, identify it as a helper rather than pretending it is part of total.py. The file map should always make clear which program calculates the total and which tool merely examines an existing result.

18 Separate supported inputs from accidental acceptance

A documentation author often discovers that a small program accepts more input shapes than the project intends to support. The right response is not to conceal those observations or promise that every accepted shape is meaningful. In this project the header is strict, but row validation is narrow. The supported instructional contract can require two fields per row and integer quantities, while a limitation explains that the implementation does not fully enforce row shape. That statement helps a reader supply appropriate data without claiming protections that do not exist.

Work through three contrasts. A header-only file has the exact required header and no data rows; the sum is zero, and the script creates a zero-total report. A duplicate-ID file contains two quantity rows that happen to share an identifier; both quantities contribute because the identifier is unused by the arithmetic. An extra-field row contains an additional trailing cell; DictReader preserves that extra value separately, but the script only reads quantity. The first case is a natural boundary input, while the other two expose validation limits. All three are useful for documentation because they prevent a broad phrase such as “invalid records are rejected” from passing unexamined.

A negative quantity is another example of a distinction between computation and domain interpretation. The lot fixture includes negative two, and the program adds it normally. The result nine is correct for the implemented arithmetic. Whether a negative quantity should represent a return, a correction or invalid business data cannot be decided by this script. The README should say that signed integers are processed and that domain rules are outside scope. It should not infer a business meaning merely because a value can be converted to int.

When writing a limitation, name the missing check and its consequence. “No duplicate-ID validation; rows with the same identifier are still summed” is more useful than “some edge cases may exist.” “No full row-shape validation; extra trailing fields can be ignored” gives a reader a concrete reason to prepare the recommended input format. These statements are not invitations to use malformed data. They are boundaries that keep the tool's behavior from being misunderstood.

A reviewer can challenge a validation claim with a small counterexample rather than a long debate. The supplied duplicate and extra-field fixtures are such counterexamples. Run them only in a fresh local copy with new output names, preserve the observations, and update the README wording accordingly. Do not silently alter the program to reject the fixture, because that would answer a different question. The documentation task is to describe the actual implementation and its supported use accurately enough that another learner can reproduce both the ordinary result and the known limits.

19 Evaluate a complete handoff with an evidence matrix

An evidence matrix connects the README's promises to the observations needed to accept them. Put one promise in each row, then identify its source, test and acceptance condition. For example, the promise that items.csv produces ten is supported by the input arithmetic and a successful fresh-target run. The promise that existing outputs are preserved requires a pre-existing sentinel file and a check that its content remains unchanged after rejection. This structure prevents a single happy-path run from being stretched to cover unrelated claims.

The matrix should also reveal claims that cannot be fully established in this exercise. “Works on every machine” has no finite local acceptance test here and should be removed. “No network operation appears in the supplied source” is a source-inspection claim, not a statement that a network-monitoring experiment occurred. “The peer followed the README without help” requires an actual peer observation; a solo rehearsal can instead be described as a simulated fresh-reader test. Accurate labels preserve the value of each kind of evidence without pretending they are interchangeable.

A practical acceptance decision can have three outcomes. Accept means the essential interface is correct and the required reproduction evidence is present. Revise means a specific documentation defect prevents or misleads a reader, such as a reversed argument order or an overwrite claim. Incomplete evidence means the instructions appear consistent with the source but a required run has not happened. This third category is important: a

missing execution environment is not proof that the README is wrong, but it also does not justify a claim of successful reproduction.

Prioritize revisions by their effect on the reader. Fix commands, input assumptions and output protection before polishing phrasing. Then resolve contradictions between examples and file names. Finally, improve navigation and brevity without deleting essential conditions. This order makes review efficient because it addresses errors that can cause incorrect results or unnecessary file loss before cosmetic issues. The final document should remain short enough to use, but completeness is defined by the task, not by an arbitrary word limit.

The matrix is also useful for fair assessment. A learner who honestly reports not run should receive credit for correct predictions and transparent reporting, but not for execution evidence they did not produce. A learner who submits a plausible fabricated log should not receive that execution credit either. The goal is to reward observable competence: accurate instructions, appropriate tests, preserved data and defensible conclusions. A README becomes trustworthy when its claims, examples and evidence agree.

Promise	Evidence needed	Acceptance condition
Correct sample total	Independent arithmetic and fresh run	10 in the named new file
Existing report preserved	Sentinel before and after rejected run	Same sentinel text after error
Malformed quantity rejected	Fraction fixture with fresh target	ValueError and no target
Clear first-reader route	Clean-copy procedure or actual peer run	No hidden prerequisite or filename guess
Honest compatibility scope	Actual environment record	No unsupported universal claim

20 Complete a reader-centered handoff

The final handoff should contain enough material for another learner to start without contacting you for missing context. Include the README, the unchanged program, the intended input fixtures and the reproduction log. Keep generated examples clearly distinguished from required source files. A second learner should be able to identify the first command, calculate the expected sample result independently and understand why rerunning with the same output path fails. The quality of the handoff is measured by this observable capability, not by decorative formatting.

Review the handoff in two passes. First, check technical accuracy: names, argument order, header, total, output text, failure behavior and environment claims. Second, check usability: can an unfamiliar reader find the prerequisite, locate the correct folder, copy the command and recover safely? These passes answer different questions. Accurate facts scattered across unrelated paragraphs can still produce an unusable README, while a beautifully organized document can still contain a dangerous overwrite instruction.

Ask the reader to report every point where they had to infer an unstated step. Their questions are diagnostic evidence, not proof that they are careless. If they looked for terminal output, improve the artifact instruction. If they copied a preexisting report into the new folder, improve the file map and starting-state description. If they expected duplicate identifiers to be rejected, strengthen the validation boundary. Each improvement should address a real misunderstanding with a precise statement.

The final claim should stay proportional to the work. You can say that the supplied fixture was reproduced in the recorded environment if that happened. You can say the documented negative cases behaved as recorded if they were executed. You cannot infer production readiness, data quality certification or compatibility with every system. A modest, evidence-backed README is more trustworthy than a universal promise. The skill you are practicing is transferring a reproducible local procedure from one person's knowledge into a document another person can actually use.

E20-X1 Deliver the complete reproducible README

Create a complete README for total.py with items.csv as the main example and lot.csv as a second example. Include purpose, prerequisites, initial file map, working folder, argument order, exact output for both examples, silent stdout, fresh-output policy, input contract, troubleshooting for wrong header and fractional quantity, and limitations. Perform a clean-copy run if possible and attach your actual log, otherwise state the blocker.

Submit: README.md or README.txt plus an evidence log. Do not modify total.py or the input fixtures.

Assessment

Choose one best answer per question. Questions 1–10 check the foundation concepts; questions 11–20 apply the extended study units. Complete and submit the practical tasks separately. A strong quiz score does not replace correct practical work.

1. What is the central purpose of this project README?

A. Enable an unfamiliar reader to reproduce the supplied local task

B. Describe every Python language feature

C. Promise that all CSV data are trustworthy

D. Replace the program with a narrative summary

2. Where should the relative-path example be run?

A. In the folder containing total.py and the chosen CSV

B. In the output folder even when the script is elsewhere

C. In the Python installation folder

D. In any terminal folder because Python searches every directory

3. Which argument order matches total.py?

A. New output path, then input CSV path

B. Input CSV path, then new output text path

C. Input path only, because output is printed

D. Output directory, then the expected total

4. For quantities 2, 5 and 3, which output line is expected?

A. Total quantity: 235

B. Total quantity: 3

C. Total quantity: 5

D. Total quantity: 10

5. Why must the README explain that a successful run can be silent?

A. The normal result is written to a file, not standard output

B. Silence proves that no file was created

C. The program requires a second run before displaying success

D. Errors are suppressed by the script

6. A documented output path already exists. What is the safest routine next step?

A. Switch to write mode in the supplied source

B. Delete every text file in the folder

C. Choose a different absent output filename

D. Use the input file as the target

7. Which prerequisite statement matches the supplied project?

A. Install a CSV package before every run

B. Create a virtual environment because pathlib is third-party

C. Use an already available Python 3.10+; the script uses only its standard library

D. Configure a service account to store the result

8. What does a clean-copy test primarily reveal?

A. Whether every later Python release will work

B. Whether the instructions depend on hidden files or remembered setup steps

C. Whether the output can be published automatically

D. Whether the input represents real inventory accurately

9. Which record is strongest evidence of a reproduction run?

A. The expected command copied into a log with no result

B. A screenshot containing an old report with no command

C. Actual interpreter, command, result and output-state observations

D. A statement that the README looks professional

10. Which statement overstates the evidence?

A. The header check expects id,quantity in that order

B. The program is guaranteed to work in every environment

C. The recorded local run used the stated interpreter

D. The fixture quantities sum to ten

11. The valid header is present but there are no data rows. What does this implementation write?

A. Total quantity: 1 because the header is counted

B. No file because empty input is always rejected

C. Total quantity: 0 followed by a newline

D. The previous output total

12. Why is the sentence “duplicate IDs are rejected” inaccurate?

A. DictReader automatically rejects repeated id values

B. A correct total proves all source identifiers were unique

C. The exact header comparison also validates each record identifier

D. The code does not compare or track identifier values

13. What distinction should a log make between expected and observed results?

A. Expected results are facts about the learner computer

B. The columns are interchangeable after writing the README

C. Observed results may be filled from the sample answer

D. Expected results are predictions; observed results require an actual run

14. A quantity cell contains 2.5. Which explanation matches the program?

A. The value is rounded up

B. The value is truncated to two

C. Integer-string conversion fails before a new output is opened

D. The row is skipped while the remaining total is written

15. What can a successful extra-field.csv run establish?

A. Every malformed CSV is accepted safely

B. The code may ignore an extra trailing row field; it does not enforce complete row shape

C. The third field is included in the numeric total

D. The header comparison allows arbitrary extra columns

16. Which evidence best checks the existing-output protection?

A. A nonzero return code alone

B. A new empty report after the error

C. A different report created by a later command

D. The expected FileExistsError and unchanged pre-existing target content

17. Why should the README avoid demanding a fixed traceback path?

A. Traceback paths are never useful in debugging

B. A fixed path guarantees the correct input

C. Python errors contain no file information

D. Paths and line details can vary without changing the documented failure category

18. What is the main limitation of using only rehearse.py to validate the README?

A. It checks program cases but cannot prove that a human can follow unclear prose

B. A subprocess argument list also tests every shell quoting convention

C. The runtime recorded by the runner establishes compatibility with every Python release

D. Passing program cases proves that every README command names the right folder

19. The lot.csv example replaces items.csv. Which linked facts need review?

A. Input name, command, output name and expected total together

B. Only the interpreter brand

C. Only the README title

D. Only the length of the log

20. A suitable interpreter is unavailable. What is the honest reproduction-log entry?

A. Passed because the source looks correct

B. Not run, with the missing prerequisite recorded

C. Passed using the model environment version

D. Failed arithmetic because ten was not observed

Sources and further reading

GitHub Docs: About READMEs
https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes

Python: Input and output
https://docs.python.org/3/tutorial/inputoutput.html

Python built-in functions: open and int
https://docs.python.org/3/library/functions.html#open

Python csv module
https://docs.python.org/3/library/csv.html