Human-Led AI Coding

Build with AI. Stay in Control.

A free, account-free course from ViridianMint Publishing.

Revision 2026-11-09-r2

Four lessons, a worked example, practice and reference answers, and reusable resources. Estimated careful reading: 45-60 minutes; practical work takes longer.

Contents

Human-Led AI Coding

The Human Is in Charge

Giving AI a Clear Assignment

Verifying the Work

Keeping What You Build

Practice and Reference Answers

Download and Keep

AI Worker Assignment Template

Verification and Acceptance Checklist

Recovery Card

The HTML course is the primary accessible version. This PDF has selectable text and is not claimed to meet a document accessibility certification. The example is an authored teaching scenario; no real AI session or owner acceptance is claimed.

Human-Led AI Coding

Build with AI. Stay in Control.

You describe a change. An AI coding assistant produces something that looks finished. It explains the design, lists the files it edited, and says the tests pass. Now comes the part that determines whether you own the project: deciding what to believe, what to inspect, and whether the result actually serves your purpose.

This free course teaches a repeatable way to make that decision. You will follow a small change from intention to a recoverable result. The method works with an assistant in a chat window, a coding editor, or a terminal. The examples use Python and Git, but the central decisions belong to you regardless of the tools.

What you will be able to do

- Define a small change in terms of observable behavior.
- Give an assistant a bounded assignment with explicit acceptance checks and stop conditions.
- Separate a worker's claims from evidence you can inspect and checks you can rerun.
- Distinguish technical correctness from your acceptance of the experience.
- Keep the source, decisions, instructions, and recovery materials needed to continue without the original chat or service.

You do not need an AI subscription, account, email address, or purchase to use this course. You can complete the evaluation exercise by reading the supplied code and evidence. Running the example is optional. It uses Python 3 and the standard library, with no package installation or network connection.

How to use the course

Read the four lessons in order the first time. Each lesson develops one stage of the same reading-list example, then gives you a small practice task. Keep your answers in a plain text file or notebook. At the end, evaluate another candidate and compare your reasoning with the reference answers.

Allow approximately 45-60 minutes for a careful read including the code and evidence. This is a planning estimate, not a timed requirement. Hands-on exercises, running commands, and reviewing your own project take additional time. Pause whenever you need to; there is no enrollment, progress record, or deadline. Bookmark any lesson to return directly to it.

The complete course is available as Markdown and PDF. The portable ZIP also contains editable templates and the runnable example. The HTML pages are the primary accessible version. The PDF has selectable text, but is not presented as a certified accessible document.

The example you will follow

Imagine a small tool that cleans a reading list. It should remove blank entries and duplicate titles while keeping the order you chose. That sounds easy. An assistant might nevertheless alphabetize the list, change the spelling, or add a dependency you never requested.

We supply a starter, a deliberately flawed candidate, a correction, and tests. This is an authored teaching scenario. The worker reports are simulated; the recorded program outputs and test results are produced by running the supplied files. They are not evidence that a real AI session or a real owner's acceptance occurred.

The small size is useful. You can see the entire change and inspect its consequences. The same questions apply when the change is a search feature, a spreadsheet import, or a game mechanic: Who decided what success means? What changed? What supports the claims? Can you recover?

About ViridianMint

ViridianMint Publishing makes independent books and software for practical study and creative work. Our engineering approach emphasizes human judgment, bounded delegation, evidence, and durable ownership. We want a project to remain understandable and usable when an assistant, subscription, or hosting arrangement changes.

This course explains that approach in a form you can adapt. It stands on its own. You do not need to buy a product to complete it, and completion does not constitute professional certification. The goal is a useful skill: directing assistance while keeping the authority and materials needed to continue your own work.

The Human Is in Charge

By the end of this lesson, you should be able to separate the decisions you own from work you can delegate, describe a small change in observable terms, and decide how much verification it needs.

Start with the outcome

An assistant can help write code, explain an unfamiliar function, draft tests, and prepare documentation. Those abilities do not tell it which tradeoffs you accept or whether the project is worth building. You remain responsible for its purpose and for decisions made under your authority.

Write the intended outcome before requesting an implementation. Avoid beginning with a list of components. "Add a database and a dashboard" describes a proposed solution. "Let me find yesterday's orders without reading every row" describes a problem you can evaluate. A solution may follow, but first establish what the person using it needs to accomplish.

For the reading-list tool, the purpose is to remove repeated entries without undoing the reader's choices. The order may represent a study plan. The capitalization may be how the reader recognizes a title. Alphabetical order would be a plausible feature in another tool, but it would damage this use.

Our first-use statement is simple: a reader passes a list of titles to a function and receives a clean list in the same chosen order. The source list is preserved. No file is overwritten, and no network or account is involved. That gives us a concrete experience before we choose an algorithm.

Keep an authority boundary

You can delegate implementation without delegating the right to change the project. For a small assignment, state who may edit files, introduce dependencies, commit changes, publish, or spend money. These permissions are different. Access to a terminal does not itself establish permission to release software.

In this course, the owner chooses the behavior and acceptance criteria. The worker may propose and implement a bounded change in an isolated candidate. The reviewer inspects the change and reruns checks. The owner evaluates the result and decides whether to accept it. A real project may involve one person doing several roles, but the decisions still need to be distinguishable.

An independent reviewer has not authored the change being judged. A second assistant can provide another inspection, but it can share assumptions with the first. Its report is additional evidence, not a guarantee. For consequential work, use checks that can contradict both assistants: actual data, separately chosen cases, or a person with relevant expertise.

You do not need a large team to practice this. As an individual, you can ask for a candidate, close the worker's summary, and inspect the source against your own written criteria. Describe that honestly as an owner review, not an independent third-party review.

Turn the intent into a small contract

Here is the behavior we will use throughout the course. The function accepts a list of strings and returns a new list of strings. It trims surrounding whitespace, omits entries that become empty, and treats titles as duplicates using Python's Unicode casefold() comparison. It keeps the first retained spelling after trimming and preserves the order of first occurrence. It does not change the input list.

Using casefold() is a precise choice for this example. We are not claiming that it solves every language or cataloging problem. We do not combine internal spaces, remove accents, compare punctuation, or normalize Unicode forms. "Part One" and "Part One" remain distinct: Part[space]One has one space; Part[space][space]One has two. The brackets label spaces, not literal text. If a future user needs different matching rules, that is a new product decision.

The accepted input type is also bounded. We are not designing behavior for numbers, nested lists, or a damaged text file. A function for trusted strings needs less infrastructure than a public upload service. Naming this boundary keeps an assistant from adding speculative error handling or quietly inventing semantics.

Write one example before any code:

```
titles = ["  Dune  ", "", "Earthsea", "dune", "  ", "EARTHSEA", "Solaris"]
expected = ["Dune", "Earthsea", "Solaris"]
```

The example communicates several rules together. It does not replace the written contract: a candidate could happen to pass this list while mishandling another order, Unicode text, or the empty list. We will use additional cases later.

Give the assistant work you can evaluate

Tasks become difficult to supervise when they combine unrelated decisions. "Finish my app" may include product design, file layout, deployment, security, and pricing. You cannot easily tell whether the assistant made an acceptable change or chose a new direction.

A bounded task has a coherent purpose and an end you can recognize. "Implement the agreed reading-list behavior in this function, with tests and a short handoff" is small enough to inspect. It still requires judgment. Bounded does not mean reducing every action to a separate prompt or micromanaging every variable name.

Let the worker choose routine details within the contract. A set for seen comparison keys is a normal implementation choice. Changing the duplicate rule, adding a package, or writing back to a user's file changes the agreement and should come back to the owner. The distinction lets you preserve agency without supervising every keystroke.

Match verification to consequences

Consider what failure would affect, how quickly you would notice, and how you would recover. A local function in a disposable example needs focused checks. A change to a shared file format needs inspection of old data and compatibility. A payment or access-control change requires broader evidence and appropriate expertise before release.

The reading-list example is deliberately low impact. We can inspect its complete source, run representative tests, and compare a hand-worked case. We do not need a cloud testing service, a large benchmark, or a simulated production rollout. Adding those would consume attention without resolving the actual uncertainty.

Risk can change during work. If the worker discovers that the function is also used by an importer that deletes records, stop and reassess. A task that looked local may touch persistent data. Do not keep using the original verification plan after learning that its assumptions were wrong.

When you cannot judge the code yet

You can still ask useful questions. Ask what the function receives and returns. Ask which exact rule each test covers. Ask the worker to show a small trace: what happens to one title, what key it computes, and when it appends an entry. Compare the explanation to the actual lines rather than accepting a paraphrase alone.

For unfamiliar or high-impact changes, make learning and review part of the task. An assistant's ability to explain itself does not make its answer correct. A useful explanation helps you identify a check you can perform or a question for a qualified reviewer. If the important decision remains beyond your ability to assess, keep the candidate out of production until that gap is addressed.

Practice: write your owner's note

Choose a change small enough to describe in four sentences. State who uses it, what they do first, what result they should observe, and what must remain unchanged. Add one realistic failure and its consequence. Keep implementation details out unless they are a genuine constraint.

For our example, a sound note might say: "I choose the reading order. The tool cleans repeated titles while keeping that order and my first spelling. It returns a new list, leaving my input untouched. A sorted result is unacceptable because it destroys my study sequence."

Compare your note with that structure. Can another person tell when the experience succeeds? If the only criterion is "works well," make it more observable. Carry the note into the next lesson; it will become the reason behind the worker's assignment.

Giving AI a Clear Assignment

By the end of this lesson, you should be able to turn an intent into a bounded assignment, choose acceptance cases before implementation, and write stop conditions that prevent an assistant from guessing through a material decision.

Give context that changes the work

An assistant needs the relevant source, constraints, and evidence. It does not need your entire project history. Provide the specific files and current baseline, describe the intended behavior, and identify anything the worker must preserve. If you have local instructions, include them in the repository so future work can find them.

For our example, the starting file contains only a function with a docstring and NotImplementedError. The supported input is a list of strings. There is no user interface, file import, package manager, or deployment. That context prevents the worker from turning a small function into an application.

In a real repository, ask the worker to inspect the branch and working-tree state before editing. If it finds unrelated work, it should preserve it and describe the overlap. A clean baseline makes a change easier to compare, but cleanliness is not permission to reset a repository. Never have an assistant discard files merely to make the starting state convenient.

The assignment we will use

Here is a complete example assignment. It is a teaching artifact, not a claim that a real assistant received it.

```
Objective: Clean repeated reading-list entries without changing the reader's order.

Baseline: supplied starter/cleaner.py in a disposable example directory.
Interface: clean_titles(titles), where titles is a list of strings.

Required behavior:
- Return a new list; leave the input unchanged.
- Trim surrounding whitespace and omit blank entries.
- Compare duplicates with str.casefold().
- Preserve the first retained spelling after trimming.
- Preserve the order of first occurrence.

Scope: implement cleaner.py, add focused standard-library tests,
and write a short verification and recovery note.

Exclude: sorting, internal-space normalization, file I/O, networking,
new packages, interfaces, Git commits, publishing, and unrelated edits.

Acceptance: representative examples and all supplied contract cases pass;
the diff is limited to the allowed files; reviewer checks input preservation.

Stop: if the interface or starting source differs, a non-string policy is
needed, dependencies seem necessary, or required behavior conflicts.
Report the issue and propose a bounded resolution; do not silently expand scope.

Handoff: exact changed files, readable diff, commands and exit statuses,
failures or unrun checks, remaining limitations, and recovery instructions.
Do not claim owner acceptance.
```

This assignment leaves the algorithm open. The worker can use a loop and a set without requesting approval. It cannot decide to sort because sorting changes the product. It cannot add a web form because there is no authorized interface work. The difference is whether a choice implements the agreement or replaces it.

Choose cases that distinguish correct from plausible

Acceptance criteria should describe behavior you care about, including cases where a plausible implementation would be wrong. A test with already sorted lowercase titles cannot tell you whether the function wrongly sorts or lowercases everything. Choose inputs that expose those transformations.

Use ["Solaris", "Dune", "Earthsea"] to check order. Use ["Dune", "dune"] to check first spelling. Use spaces and tabs to check trimming. Use [] to check the empty list. For this contract, ["Straße", "STRASSE"] checks the declared casefold() comparison rather than simple lowercase matching. These examples are selected from the requirement, not from the code the worker happens to produce.

Also test what should remain distinct: ["Part One", "Part One"] should keep both titles because internal whitespace normalization is excluded. Respectively, these are Part[space]One (one space) and Part[space][space]One (two); the brackets label spaces, not literal text. Negative boundaries like this prevent well-intentioned expansion from hiding inside a "cleanup" change.

Keep tests readable. A newcomer should be able to compare the input and expected result without understanding a custom testing framework. Standard-library unittest is sufficient here. A small suite with meaningful cases is more useful than dozens of assertions that repeat the same easy example.

Ask for evidence in a usable form

"Tests pass" is a claim. A stronger handoff names the command, where it was run, the candidate it tested, and the exit status. It includes failures and checks that were not run. It gives you enough source and instructions to repeat the result without relying on the worker's account.

For a real assignment, request a diff against the starting point. A file list alone cannot show that the worker also changed a setting or rewrote a test. A readable diff reveals deletions, added dependencies, changed assertions, and accidental formatting churn. You can then focus on behavior rather than searching a whole tree for edits.

Do not ask for a polished story that suppresses uncertainty. "The example passes, but the input-mutation check was not run" is useful information. "Fully tested and production ready" is less useful if the evidence covers one toy case. Specify the evidence you need and let the result remain incomplete when it is incomplete.

Use stop conditions for decisions, not routine friction

A stop condition identifies a situation where the worker should return control. It should be specific enough to recognize. "Stop if something goes wrong" is vague. "Stop if the function also writes to a saved library" identifies a change in risk and authority.

Avoid making every implementation detail a stop. If the worker can correct a typo or choose a variable name within the assignment, it should proceed. Good stop conditions preserve your attention for scope, data, dependencies, unclear behavior, and irreversible actions.

When a stop occurs, ask for the concrete conflict and a proposed resolution. You might narrow the task, approve a changed boundary, or reject the approach. Record the revised agreement before work continues. An endless sequence of guesses is not progress, even if every guess produces more code.

Revise assignments using evidence

Suppose you forgot to specify whether duplicate comparison ignores case. The worker asks, and you decide it should. Add the rule and a distinguishing test. Do not leave that decision only in a passing chat message. It belongs with the assignment and tests so the next worker can see it.

If a worker repeatedly misreads a requirement, inspect the wording before escalating the prompt's tone. "Clean the list" could reasonably mean sorting, standardizing spelling, or removing punctuation. A sharper behavioral contract often resolves the problem more effectively than adding general demands to be careful.

Long prompts are not inherently more precise. Remove constraints that do not affect this task. Keep the purpose, allowed scope, acceptance, stop conditions, and handoff easy to find. The editable assignment template in the downloads provides those fields without requiring you to adopt a particular coding assistant.

Practice: inspect the simulated first handoff

The worker returns this candidate and says, "Removed blanks and duplicates; the sample run completed without errors." The report is simulated. Read the implementation before looking at the next lesson's evidence.

```
def clean_titles(titles):
    return sorted({title.strip().lower() for title in titles if title.strip()})
```

List two ways this could violate our assignment while still returning a valid Python list. Then choose a test that makes each violation visible. Resist the urge to rewrite the whole task. A correction request should name the failed requirement and preserve the accepted scope.

A strong answer identifies changed order and lost first spelling. A further issue is using lower() instead of the agreed casefold() comparison. We will run those cases, separate the evidence from the report, and decide what acceptance should mean next.

Verifying the Work

By the end of this lesson, you should be able to evaluate a worker's handoff, select checks from the requirements, distinguish different kinds of evidence, and make a reasoned accept-or-revise decision.

Start with the change, then the explanation

An assistant's summary is useful for orientation, but it is not the source of truth. Open the diff and compare it with the assignment. Identify the files changed, the behavior introduced, and any changes to tests or settings. If the summary says one file changed and the diff shows five, resolve that discrepancy before accepting the story.

For the reading-list example, the complete rejected implementation fits on two lines. It strips whitespace, uses a set, lowercases titles, and sorts the result. Every operation is visible. The set removes repeated lowercase values, but the sorting discards the reader's chosen order. Lowercasing changes the spelling that should have been preserved. It also does not implement the agreed Unicode comparison rule.

None of those defects requires a crash. The function returns a list and looks concise. This is why "runs without errors" is too weak an acceptance criterion. Correctness depends on the agreement, not merely on whether the programming language accepts the code.

Compare expected and observed behavior

Consider this focused case:

```
titles = ["Solaris", "Dune", "Earthsea", "dune"]
expected = ["Solaris", "Dune", "Earthsea"]
observed_rejected = ["dune", "earthsea", "solaris"]
```

The rejected output reveals two product failures at once: the order changed, and the first spelling disappeared. Use separate focused tests as well, so a failure message tells you which rule broke. A combined example shows the whole experience; isolated cases help explain the cause.

The supplied example runner executes eleven contract tests against each implementation. The deliberately rejected candidate fails five tests. The corrected candidate passes all eleven. Those are the observed outcomes for these specific files and cases; they do not establish that every possible input has been proven correct.

The complete recorded outputs are in the downloadable example. To reproduce them after extracting the portable package, open a terminal in that package's root and run:

```
python3 example/run_checks.py rejected
python3 example/run_checks.py corrected
```

The first command should finish with a nonzero exit status because it tests a known rejected implementation. The second should finish with exit status zero. Neither command changes your project or connects to the network. Python must already be installed; do not paste commands into an unfamiliar project directory and assume the files are the same.

Separate four levels of confidence

First, there is a worker claim: "I ran the tests." Second, there is recorded evidence: the command and its output. Third, there is a check you or another reviewer actually rerun against the candidate. Fourth, there is product acceptance: the owner deciding the result serves the intended experience.

These levels can support one another without becoming interchangeable. Recorded output might belong to an older file. A rerun might use tests that omit an important rule. Passing tests might still produce an awkward interface. A happy owner might have tried only the easiest path. Name what each result establishes and what remains uncertain.

For our tiny function, useful additional inspection includes checking that the input is never modified, that the set stores comparison keys rather than output spellings, and that no hidden file or network work has been added. Rerunning the same assertion many times adds little if the missing question is whether the requirement itself was understood.

Read the correction as an explanation of behavior

Here is the corrected implementation:

```
def clean_titles(titles):
    result = []
    seen = set()
    for title in titles:
        cleaned = title.strip()
        if not cleaned:
            continue
        key = cleaned.casefold()
        if key not in seen:
            seen.add(key)
            result.append(cleaned)
    return result
```

Trace one duplicate. The first Dune becomes the comparison key dune; the original trimmed spelling is appended. The later dune has the same key, so nothing is appended. Trace a new title: it is appended at the moment it first appears, with no later sorting. The loop reads the input and constructs separate containers, leaving the original list untouched.

This trace connects lines to requirements. It is more useful than saying the implementation uses an "efficient algorithm." Performance claims would need appropriate measurements if performance mattered. For the tiny supplied lists, the important issue is whether the behavior is correct and understandable.

Inspect the tests too

Tests are code written by someone, often the same assistant that wrote the implementation. Inspect their expected results. If a test expects sorted lowercase titles, it can pass while encoding the exact mistake you want to reject. A green result is only as relevant as the question the test asks.

The supplied suite covers empty input, whitespace-only input, trimming, ordinary duplicates, mixed-case duplicates, first spelling, unsorted order, Unicode case folding, input preservation,

return of a new list, and preservation of internal whitespace distinctions. The rejected candidate passes some of these. That matters: an honest report must show the failures as well as the successes.

For a larger change, inspect boundaries between components and old data as well. A unit test for one function does not demonstrate that an application calls it correctly. A browser screenshot does not demonstrate that a download contains the right bytes. Choose evidence at the level where the consequence appears.

Write a bounded correction request

A useful response to the rejected candidate is: "Revise. The result sorts the list and loses the first spelling. It also uses lowercasing instead of the agreed casefold comparison. Preserve original order and first trimmed spelling, add distinguishing tests for those rules, and rerun the full suite. Keep the existing interface and exclusions."

That request identifies the contract failures and the evidence needed to close them. It does not add a new interface or silently expand the task. If you find yourself changing the intended behavior instead, acknowledge a product revision and update the agreement before asking the worker to continue.

Avoid vague rejection such as "make it better." It leaves the assistant to guess what bothered you and may trigger unrelated changes. Avoid accepting a promise to fix it later when the failed behavior is part of the current task's definition of success.

Make an acceptance decision with limits

For this example, an engineering review can conclude that the corrected source satisfies the specified cases, stays within scope, and preserves input. It should still identify the supported input type and the exact matching rule. Those limitations are part of the interface, not unfinished features to conceal.

The owner then tries a representative personal list. Does the result preserve the intended study sequence and recognizable spelling? If it does, the owner may accept the behavior. This course does not supply a fabricated owner acceptance. Your exercise is to explain what you would check and why your decision follows from the evidence.

In your own project, write a short acceptance note: the candidate identifier, requirements reviewed, checks actually performed, limitations, decision, and who made it. A decision with a date and source reference is easier to audit than a chat message saying "looks good" that cannot be connected to the final files.

Practice: evaluate one missing case

Suppose the corrected implementation passed every output check, but nobody compared the input before and after the call. Would you accept an assignment that explicitly requires input preservation? Identify the remaining uncertainty and propose the smallest check that resolves it.

A sound answer is to copy the input list, call the function, and compare the original with the copy. Also inspect whether the function sorts or edits the input in place. The missing evidence does not automatically prove a bug. It establishes a question that must be resolved before the acceptance criterion is met.

Keeping What You Build

By the end of this lesson, you should be able to identify the materials needed for durable ownership, use a cautious review-and-commit sequence, and design a recovery check that does not depend on the original chat.

Ownership needs usable materials

Having a running program is only part of owning a project. You also need the source that produced it, instructions for building or running it, dependencies and their versions, relevant decisions, verification evidence, and a way to recover an earlier accepted state.

An exported binary without source may be usable today but difficult to repair. A repository without setup instructions may be hard for another person to build. A long chat may explain intentions while leaving the actual accepted state unclear. Preserve a small, coherent record in the project itself.

For our example, the handoff is compact: the function, tests, behavior specification, rejected and corrected candidates, exact commands, recorded results, a readable diff, and a README. Someone can extract the package, run the checks offline, and understand why the rejected result was unacceptable. None of that requires access to the assistant that helped create it.

Keep the project understandable from the outside

Write a README for a future maintainer who has not seen your chat. Explain what the tool does, its supported inputs, how to run it, how to test it, and where its boundaries are. State whether setup downloads anything and which files a command writes.

A useful dependency record distinguishes runtime needs from development tools. The example needs Python 3 and the standard library to run and test. It does not need ReportLab; that is a development tool used to build this course's PDF. Mixing those requirements would make the example appear harder to use and less portable than it is.

Record the reason behind decisions likely to be questioned. "Preserve first occurrence because order represents a study plan" explains why sorting is a defect. "Use casefold for comparison; do not normalize internal spaces" defines a deliberate boundary. You do not need to narrate every discarded variable name or intermediate typo.

Keep this record close to the source. A later worker should discover it through the repository's ordinary files. Documentation that only one person can locate in a private chat is fragile continuity.

Use version control as a decision record

Git records snapshots and their history. It helps you inspect changes, group an accepted change coherently, and identify a prior state. It does not automatically establish that a commit is correct, safe, or complete. Review comes before the commit, and recovery depends on preserving the repository and its history.

The following commands are Git-specific examples. Run them only inside a repository you recognize. The first commands inspect; they do not alter the working files:

```
git status --short --branch
git diff -- cleaner.py tests/test_cleaner.py README.md
git diff --check
```

status shows the branch and tracked/untracked changes. diff shows unstaged edits in the named paths; it does not show the contents of new untracked files, which you must open separately. diff --check checks whitespace errors, not program correctness. Read the output and resolve unexpected files before staging anything.

After you have reviewed and accepted the intended files, stage only those paths. Staging changes Git's index, not your working file contents. Inspect the staged change before committing:

```
git add -- cleaner.py tests/test_cleaner.py README.md
git diff --cached --stat
git diff --cached
git status --short --branch
git commit -m "Implement order-preserving reading-list cleanup"
```

These paths are illustrative for a small project laid out that way; the course package has separate candidate directories. Do not paste this sequence into a repository with different paths. The owner executes the commit after confirming that the staged diff contains only the accepted task. If other changes were already staged, explicit paths alone do not remove them from the eventual commit. Inspect the entire staged diff.

A commit creates a local history entry. It does not publish the project or make an off-device backup. A push is a separate action that sends commits to a configured remote. Deployment is another separate action. Do not treat permission for one of these as permission for all three.

Preserve unrelated work

Real repositories are often busy. A document may be half edited, another feature may be staged, or a local configuration may be intentionally untracked. Ask for a status inspection and a bounded change inventory. If the assignment overlaps existing work, stop and reconcile it with the owner.

Do not use a destructive reset or broad cleanup to make the diff easier to read. Do not let a worker overwrite a whole repository from a ZIP when only a few files should change. Work in a separate copy when that is the safest way to prepare a candidate, then integrate the precise reviewed delta.

Isolation has limits. A copied folder may still contain scripts that act on absolute paths or shared services. Inspect commands before running them. In this course, the example runner only loads a local candidate and executes tests; it neither writes your reading list nor contacts a service.

Design recovery before release

A recovery plan answers what state you want back, which materials restore it, who authorizes restoration, and how you will verify the result. "We can undo it" is not enough if the previous build, data, or credentials are missing.

For a source-only change, preserve the baseline and an exact diff. Test applying the diff and reversing it in a disposable copy, then compare the restored files with the baseline. That demonstrates a narrow recovery path without touching the owner's real work. It does not establish recovery of a production database, user uploads, or an external account.

For the reading-list tool, recovery is especially simple because it does not persist data. Keep the starter and corrected files, identify the accepted candidate, and run the same tests after restoring the intended version. If a later version writes files, backup and recovery requirements must expand with that consequence.

Git's history can support recovery, but choosing a recovery command depends on whether changes are committed, shared, or mixed with unrelated work. Do not paste a generic reset command from a tutorial into a real incident. Inspect the state and preserve a copy first; obtain approval for destructive or consequential actions.

Check portability with a fresh start

A practical portability test is to extract or clone the project into a new directory and follow only its README. Can you run the tests without the original working folder, hidden packages, assistant account, or chat history? Does the documentation distinguish optional tools from required ones?

For this course's ZIP, a reader can use the combined Markdown and PDF offline. An optional local run uses python3 example/run_checks.py corrected from the extracted root. The README explains the expected failure of the rejected candidate so a nonzero result is not confused with a damaged download.

Checksums help detect unintended byte changes when compared with an expected manifest. A checksum bundled with a download is not proof of who authored it: an attacker able to replace both could replace them together. Use it for integrity within the stated trust boundary, and do not confuse that with a signature or a guarantee about code behavior.

Practice: write a recovery card

Write five short lines for your own small change: current accepted baseline, candidate identity, verification command, materials needed to restore the baseline, and the check that confirms restoration. Add who must authorize the action if it affects shared or published work.

Now imagine that your assistant service disappears tomorrow. Could you still understand the result, run the checks, and explain the main decision? If not, identify the smallest missing record and add it to the project. Durable ownership is built through those concrete materials and practiced recovery, rather than confidence in a particular assistant.

You have now followed the full cycle: define the experience, delegate a bounded change, inspect and verify the result, make a human acceptance decision, and preserve the project. The next page gives you evidence to evaluate on your own.

Practice and Reference Answers

This exercise is an authored scenario. You can complete it without Python, Git, or an AI account. The candidate code and observed outputs below provide the evidence you need. Optional runnable files are included in the portable package.

Your assignment as reviewer

Use the same reading-list contract: trim edges, omit blanks, compare duplicates with casefold(), preserve the first trimmed spelling and original order, return a new list, and leave the input untouched. Supported inputs are lists of strings. No packages, file writes, network access, or interface changes are authorized.

The worker is asked to correct the rejected candidate. It submits this patch, provided in full in the downloads:

```
-    return sorted({title.strip().lower() for title in titles if title.strip()})
+    result = []
+    seen = set()
+    for title in titles:
+        cleaned = title.strip()
+        if not cleaned:
+            continue
+        key = cleaned.lower()
+        if key not in seen:
+            seen.add(key)
+            result.append(cleaned)
+    return result
```

The simulated handoff says: "Corrected order and first spelling. All three smoke tests passed. I recommend acceptance." The smoke cases are empty input, whitespace trimming, and an ordinary mixed-case duplicate. The worker gives no evidence for the Unicode comparison rule.

Supplied observations

The smoke runner exits with status zero. A fuller run against the supplied eleven-case contract suite exits with status one: ten tests pass and the Unicode case-folding test fails. These results are produced by running the exercise candidate; the report above is a simulated worker claim.

```
input_titles = ["Straße", "STRASSE"]
expected = ["Straße"]
observed_exercise = ["Straße", "STRASSE"]
```

For a second check, ["Solaris", "Dune", "solaris"] produces ["Solaris", "Dune"], and the input list remains unchanged. The candidate has no imports and no file or network calls. Those observations support some requirements while leaving the failed comparison rule unresolved.

Write your review

1. Decide whether to accept or request revision under the existing contract. Explain the deciding requirement.

2. Identify what the smoke-test evidence establishes and what it does not establish.
3. Name the smallest correction and the checks you would require afterward.
4. Write a bounded correction request in three or four sentences.
5. State who decides product acceptance and who may commit or publish.
6. List the materials you would keep so someone else can reproduce the decision offline.

Take a moment to answer before opening the reference section. You are practicing a decision from evidence, not guessing which wording a quiz expects. Equivalent sound reasoning is welcome.

Reference answers

Decision: request revision. The agreed duplicate comparison is casefold(). The candidate uses lower(), which does not merge the supplied Straße and STRASSE case. A successful smoke run cannot override a failed explicit acceptance criterion. The candidate is a useful improvement, but is not yet complete under this assignment.

Evidence limits: the three smoke tests demonstrate the stated results for those cases and this candidate. They support trimming and ordinary duplicates, but do not cover the declared Unicode comparison. The fuller run identifies the actual missing behavior. The absence of imports and the inspected function body also support the bounded scope; they do not certify the safety of a larger application.

Smallest correction: change cleaned.lower() to cleaned.casefold(). Keep the stored output as cleaned so the first spelling is preserved. Rerun the eleven-case suite against the corrected file, check the input-preservation case, and inspect the diff for unrelated changes. A larger rewrite or a new dependency is unnecessary for this requirement.

Possible correction request: "Revise the duplicate comparison to use the agreed casefold rule. The supplied Unicode case currently returns two titles where one is expected. Keep the existing interface, first spelling, original order, and scope exclusions; rerun the complete contract suite and provide the exact result plus the diff. Do not commit or publish."

Authority: the owner evaluates the desired experience and accepts the product. A reviewer supplies technical findings; a worker supplies a candidate and evidence. Under this example's permissions, the worker may not commit or publish. In another project, those permissions could be different, but they must be explicit rather than inferred from tool access.

Recovery materials: keep the assignment, baseline source, exercise and corrected candidates, tests, exact commands and results, readable diff, limitation note, and the owner's eventual decision. Include Python requirements and a README so another person can run it without the chat. Connect the decision to a specific source version or integrity manifest.

Optional local practice

After extracting the portable ZIP, open a terminal in its root. These commands execute only the included local example tests:

```
python3 example/run_checks.py exercise --smoke
python3 example/run_checks.py exercise
python3 example/run_checks.py corrected
```

Expect exit statuses zero, one, and zero respectively. The exercise failure is intentional. Compare your review with the actual output before editing anything. If you want to practice a correction, copy the example into a separate directory first; keep the supplied evidence intact for comparison.

Apply the method to your work

Return to the owner's note you wrote in Lesson 1. Fill in the assignment template for that change. Identify one plausible wrong result and a case that distinguishes it from success. Write an acceptance note that clearly separates checks you performed from checks you merely want someone to perform.

Finally, imagine handing the result to a new maintainer. Supply the source, decisions, run instructions, verification evidence, and recovery card they would need. You have completed the exercise when you can explain both your decision and how another person could check it. There is no certificate or enrollment record; keep the useful materials and adapt the method.

Download and Keep

Keep the course in an ordinary format you can read without an account. The Markdown contains the full lessons, exercise, reference answers, and editable resources. The PDF is a printable reading version with selectable text. The portable ZIP includes both, plus the runnable teaching example and its evidence.

Choose your materials

- Combined Markdown: useful for reading, searching, copying your own notes, and keeping the instructional text in a plain-text collection.
- PDF: useful for reading and printing. The HTML course remains the primary accessible version; the PDF is not claimed to meet a document accessibility certification.
- Portable ZIP: the complete offline package, including the example source, tests, recorded results, readable patches, templates, README, and integrity checksums.
- Editable worker assignment: define one bounded task, its authority, acceptance, and stop conditions.
- Verification and acceptance checklist: separate claims, inspection, actual reruns, product judgment, and release authority.
- Recovery card: record the accepted state and how to restore and check it.

The course website provides direct file links in its download panel. Downloads have no enrollment or email requirement. The revision label and source digest in the package identify the materials used to generate these formats.

Use the portable package

Extract the ZIP to a new directory. Open course.md in any text editor or course.pdf in a PDF reader. The Markdown's resource links point to files inside the package. The example README explains the optional Python commands and their expected exit statuses. No software installation is needed merely to read the materials.

The included checksum manifest detects unintended changes relative to the manifest. It is not a signature or proof of authorship. The example verifier uses only Python's standard library. Do not confuse the deliberately failing rejected candidate with a damaged download; the corrected candidate is expected to pass.

Continue independently

There is no required next purchase. Try the method on a small change whose consequences you understand. Keep your assignment, inspect the actual result, record what you verified, and preserve a recoverable state. Scale the review to the consequences as your projects become more ambitious.

AI Worker Assignment Template

Copy this text and replace the bracketed fields for one coherent change. Remove irrelevant fields rather than filling them with guesses. Keep the accepted version with the project.

Purpose and first use

[Who uses this, what they do, and the observable result.]

Baseline and context

[Repository/directory, accepted version, relevant files, supported environment. Inspect current branch and status; preserve unrelated work.]

Required behavior

[Specific rules, supported inputs, expected outputs, and what must remain unchanged. Include at least one distinguishing example.]

Allowed scope and exclusions

[Files/components that may change; excluded features, dependencies, services, data changes and incidental cleanup.]

Authority

[Who owns product decisions and acceptance. Explicit permissions for edits, installs, commits, pushes, spending, publication and external messages. Omit permissions that are not granted.]

Acceptance checks

[Cases chosen from requirements, regressions to preserve, relevant manual checks and evidence needed. Technical success and owner product acceptance are separate.]

Stop conditions

[Concrete conflicts, missing inputs, overlaps, data risks, necessary dependency changes or major design decisions that require returning control. Ask for a proposed resolution with evidence.]

Handoff and recovery

[Changed-file list, readable diff, exact commands/results, checks not run, limitations, source/setup documentation, candidate identifier, baseline and recovery check. Do not claim acceptance or independent review that did not occur.]

Verification and Acceptance Checklist

Candidate: [exact source version or manifest]

Reviewer: [name or role]; date: [date]

Scope and evidence

- Read the accepted behavior and exclusions before the worker's explanation.
- Inspect the complete diff and any new untracked files.
- Identify dependency, data, permission, and unrelated changes.
- Map meaningful cases to requirements, including plausible wrong behavior.
- Separate worker claims from recorded output and checks actually rerun.
- Record commands, environment, source identity, exit statuses, failures and unrun checks.
- Inspect tests for incorrect expectations or weakened assertions.
- Review boundaries and existing integrations according to consequence.

Human decisions

- Engineering finding: [met / revise / blocked], with reasons and limitations.
- Product evaluation performed by owner: [observations; no invented acceptance].
- Owner decision: [accept / revise / pending], connected to this candidate.
- Independent review: [who performed it and what they inspected, or pending].
- Commit/publish authority: [explicit grant or withheld]. Passing tests does not grant it.

Continuity

- Source, decisions, setup/run/test instructions and dependencies are preserved.
- Baseline and candidate are identified; recovery materials are available.
- Recovery is tested in a disposable copy where appropriate.
- No claim goes beyond the evidence actually obtained.

Recovery Card

Project and intended behavior: [description]

Accepted baseline: [version and where the complete source/history is kept]

Candidate: [source version or integrity manifest]

Run/test instructions: [commands, environment and expected results]

Recovery materials: [baseline, reviewed delta, dependencies, data backups if applicable]

Recovery action: [state-specific procedure; describe effects and preserve unrelated work]

Verification after recovery: [comparison and meaningful functional checks]

Authorization: [who approves shared, destructive or published changes]

Rehearsal: [date, disposable location and observed outcome, or not yet performed]

Unresolved limits: [missing materials, untested assumptions, external dependencies]