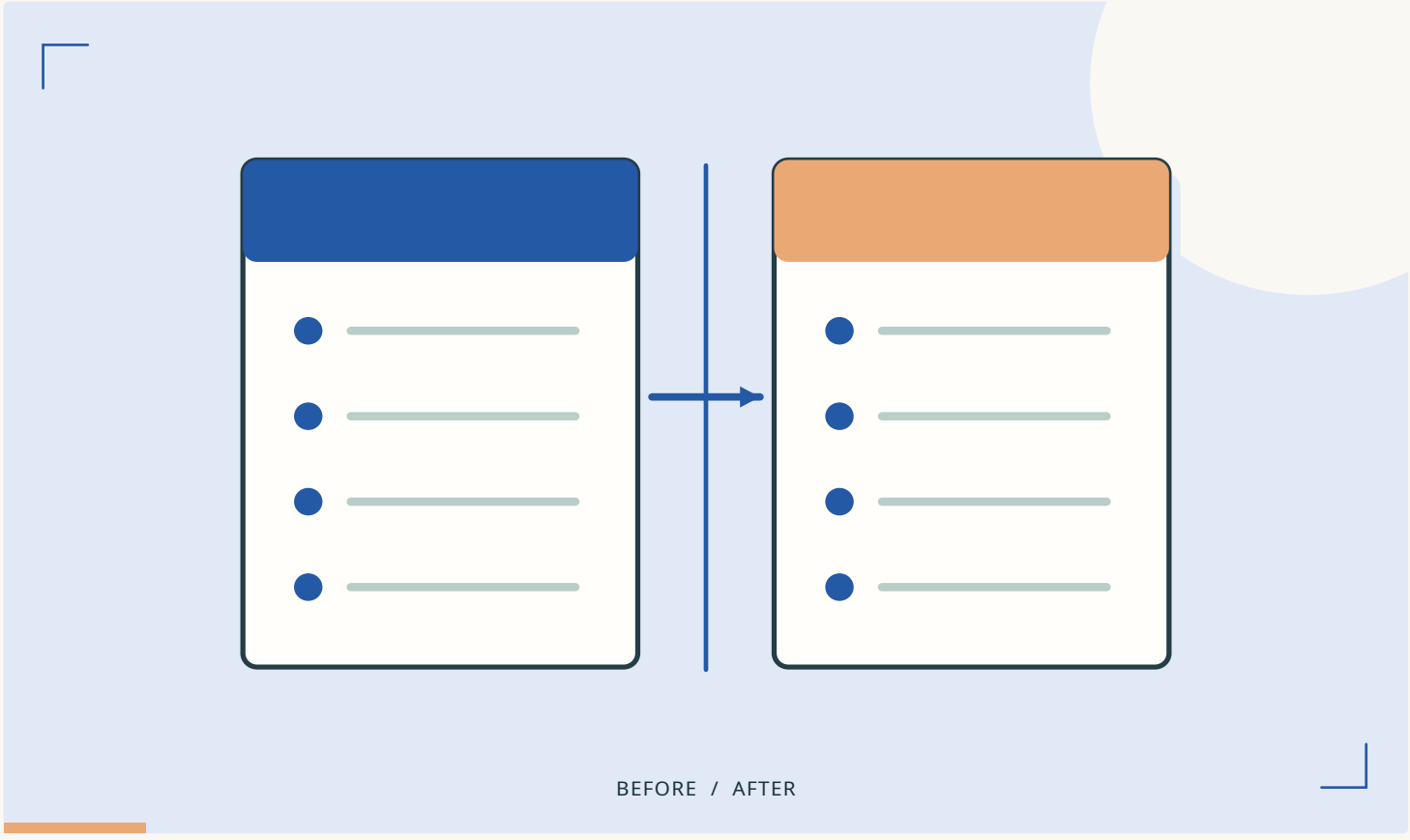
FILE DIFF REVIEW

Reading File Diffs Before Approving Changes



DIGITAL & TECHNOLOGY

Reading File Diffs Before Approving Changes

Learning outcomes

- Read additions and deletions in six small diffs.
- Separate formatting changes from behavior and requirement changes.
- Identify a removed safety condition before approving a revision.
- Reconstruct old and proposed text from unified diff metadata and content.
- Select staged versus unstaged comparisons deliberately.
- Design counterexample tests for boundaries, output preservation and cross-file contracts.
- Distinguish cosmetic edits from changed string values, control flow and usage restrictions.
- Deliver a complete evidence-led review ledger without external approval actions.

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

Read a diff as a before-and-after argument

In a unified diff, lines beginning with minus were removed and lines beginning with plus were added. Context lines orient you within the file; a changed line is often represented as a deletion followed by an addition. File headers identify old and new paths, while hunk headers show approximate line ranges. Do not count every minus line as lost functionality: it may be replaced by an equivalent plus line. Conversely, a one-character edit can change a boundary condition. Read the surrounding requirement, not just the number of changed lines.

Classify impact rather than visual size

A whitespace-only edit may be formatting, while changing >= to > excludes an exact threshold value. Removing a sentence that restricts usage can materially change a guide even when no executable code is involved. Separate behavior changes, documentation changes, configuration changes and formatting-only changes. A rename can affect scripts or instructions that refer to the old name. For each diff, state what the reader or program could observe and what test would distinguish the old version from the new one. If you lack enough context, mark the uncertainty instead of assuming that a small change is harmless.

Review against requirements before approval

The fixture contains six proposed changes, including deletion of a required no-overwrite condition. The review should flag that condition even if the remaining program still produces correct totals on the happy path. Use git diff for unstaged changes and git diff --staged for changes prepared for a commit; they show different comparisons. A review sheet should include change id, old/new meaning, impact, required check and recommendation. “Approve” means the change has sufficient evidence for the stated local exercise, not a security or production guarantee. This course ends with a review recommendation, not an actual external approval action.

Worked example

Worked example: -if quantity >= 5: followed by +if quantity > 5: changes behavior only at the threshold when quantity equals five. Tests at four and six would both produce the same results before and after; a test at five distinguishes them. A reviewer who only checks ordinary values can miss a meaningful boundary regression.

Foundation practice

Baseline requirements

- Select quantities >=5.
- Never overwrite an existing output file.
- Output filename must agree with README.
- The summary should report processed and rejected counts.

Diffs

id	file	before	after	context
C1	guide.txt	Use the kit.	Use the kit.	
C2	filter.py	if quantity >= 5:	if quantity > 5:	
C3	writer.py	if target.exists(): raise ValueError("Output exists") target.write_text(result)	target.write_text(result)	
C4	README.txt	This is a practce tool.	This is a practice tool.	
C5	settings.txt	output=result.txt	output=summary.txt	README still instructs readers to open result.txt.
C6	summary.py	print(f"Processed: {ok}")	print(f"Processed: {ok}; rejected: {bad}")	Both ok and bad are already computed and defined.

Foundation tasks

- Convert each before/after pair into an impact description, not merely “one line changed.”
- Design a distinguishing test for C2, C3 and C5.
- Capstone: deliver a six-change review sheet identifying the deleted no-overwrite condition and recommending accept, revise or clarify with evidence.

01 Decide what approval would mean before reading the patch

A diff shows a relationship between two versions. It does not, by itself, tell you whether the proposed version meets the task's requirements. The reviewer's job is to connect visible changes to observable consequences, then compare those consequences with the stated requirement. A small patch can violate an important boundary, while a larger patch can be a harmless reformatting. Counting changed lines is a navigation aid, not a decision rule.

In this course, approval means a recommendation for the supplied local exercise. You will not submit an external approval, publish a change or modify anyone else's repository. The foundation requirements are concrete: select quantities at least five, do not overwrite an existing output, keep the output filename consistent with the README, and report both processed and rejected counts. These requirements give the review a standard against which to judge the six proposals. Without that standard, a reviewer might praise a cleaner-looking program that has silently lost a required protection.

Begin a review by recording the comparison being examined, the files in scope and the requirement each file affects. Then read the proposed patch and any supplied context. Ask three different questions: what text changed, what behavior or meaning could change, and what evidence would distinguish the versions? These questions prevent the common mistake of treating a textual description as an impact analysis. “A line was deleted” describes the patch; “an existing report can now be overwritten” describes a consequence.

A recommendation should distinguish accept, revise and clarify. Accept means the available evidence supports the requirement within the stated scope. Revise means a known discrepancy needs correction. Clarify means a specific missing fact prevents a defensible decision. Clarify is not a way to avoid reasoning; name the missing fact and the observation that would resolve it. Likewise, accept is not a guarantee about every environment or every possible input. A review is strongest when its claim is both concrete and bounded.

02 Read file headers and hunk ranges correctly

A unified diff contains metadata as well as changed content. The old-file header usually begins with three minus signs and the new-file header with three plus signs. These headers identify the compared paths; they are not instructions to delete or insert literal path lines into the file. A hunk header begins with two at signs and gives old and new line ranges. Within the hunk, a space introduces unchanged context, a minus marks an old line removed from that region, and a plus marks a new line added to it.

A replacement is often displayed as a deletion followed by an addition. Read the pair together. If a line changes from quantity >= 5 to quantity > 5, the program has not lost all filtering just because a minus line appears. Instead, the inclusive comparison becomes exclusive at the threshold. Conversely, an added line is not automatically an improvement. It can introduce an undefined name, a contradictory instruction or a write mode that replaces existing data.

The line counts in a hunk include context as well as changed lines on the relevant side. In the example below, the old side contains three lines and the new side contains three lines. The unchanged function header and end comment count on both sides; the old comparison counts only on the old side and the new comparison only on the new side. These counts help locate the region but do not measure business impact. A single altered operator can still be the decisive change.

Do not infer missing context from the hunk range. A patch may show only a few surrounding lines even though a function depends on variables defined elsewhere. If the review requires that context, inspect the supplied full file or ask for the exact missing definition. The foundation C6 explicitly states that ok and bad are already defined. That fact matters. Without it, a reviewer could not justify the same confidence about the added rejected count. Read the context that is actually supplied, not the context you wish the file had.

```
--- a/filter.py
+++ b/filter.py
@@ -1,3 +1,3 @@
 def select(quantity):
-    return quantity >= 5
+    return quantity > 5
 # End of predicate
```

E02-X1 Annotate a unified hunk

For the displayed filter.py hunk, identify old path, new path, old range, new range, unchanged context, removed content, added content and the smallest distinguishing input. Explain why the two file-header lines are not removed or added program statements.

Submit: Eight labeled observations and one sentence about the behavioral consequence.

03 Reconstruct the before and after states

A diff is easier to reason about when you can mentally reconstruct both versions. Build the old version from context lines plus removed lines, and the new version from context lines plus added lines. Do not include both alternatives in one imagined program unless the actual proposed file contains both. This reconstruction prevents a subtle error: reviewers sometimes read a plus line as an extra statement that runs after the minus line, even though the minus line no longer exists in the new version.

For C3, the old code checks whether the target exists, raises an error if it does, and otherwise writes the result. The proposed code retains only the write. With an absent target, both snippets can produce the same output. With an existing target, their consequences differ. The old snippet rejects the operation under the supplied local condition; the proposed

write_text call replaces existing content. Reading only the happy path hides the removed protection because the output still looks correct in an empty directory.

Reconstruction also matters for prose. C4 replaces a misspelled word with its intended spelling. The old and new sentences have the same substantive claim, so the change is a documentation correction under the supplied context. In D3, the new sentence removes the word synthetic and broadens the usage wording. That is not merely a shorter sentence: it changes what the guide tells a reader it is for. A textual diff can therefore require the same careful meaning review as a code diff.

When a patch includes several hunks in one file, reconstruct the combined endpoint before deciding. An early hunk might rename a variable while a later hunk updates its uses, or fail to do so. Evaluating each hunk in isolation can produce either a false alarm or a missed regression. The course's small fixtures let you inspect the entire relevant endpoint. In larger work, use the diff to locate changes, then inspect enough surrounding code or documentation to establish the affected contract.

04 Choose the comparison that answers the review question

Git offers several comparisons, and a correct interpretation of the wrong comparison is still an incomplete review. In an ordinary repository with a committed baseline, git diff compares the working tree with the staging area. It shows tracked edits that have not been staged. git diff --staged compares the staging area with the current commit. It shows what has been prepared for the next commit. These commands are not synonyms, and either can be empty while the other contains important changes.

Imagine guide.txt begins as “Local practce guide.” You correct the spelling and stage that version. Then you append an unrelated note without staging it. The staged diff shows the spelling correction. The unstaged diff shows the later note relative to the staged version. A reviewer asked to inspect the proposed commit must read the staged comparison, not merely the most recent working file. If they read only git diff, they might miss the actual proposed correction and focus on a note that is not included in the commit.

Use git status --short as an inventory, then inspect the relevant content comparison. In the short format, the first status column describes index changes and the second describes working-tree changes for ordinary tracked files. MM can therefore mean a file has both a staged change and a later unstaged change. It does not mean the same change has been counted twice. The course's isolated runner demonstrates this state with synthetic files so the difference can be observed rather than memorized abstractly.

Untracked files are another blind spot. Ordinary git diff does not display their full content as tracked changes. A status entry beginning with two question marks tells you a file exists outside the index. If that file is part of the proposed deliverable, it needs an explicit review route. Do not equate an empty diff with a completely unchanged folder, and do not stage unknown files simply to make them appear. First establish the intended scope, then inspect the relevant data without expanding the task unnecessarily.

Question	Comparison or check	What it does not establish
What tracked edits are unstaged?	git diff	Contents of untracked files
What is prepared for the next commit?	git diff --staged	Later unstaged edits
Which paths have different states?	git status --short	The meaning of their contents
What differs between two named versions?	git diff OLD NEW	Which version satisfies requirements

05 Design boundary tests that can actually distinguish versions

A distinguishing test is an input for which the old and proposed versions produce different relevant outcomes. It is more informative than a test chosen merely because it is easy to run. For C2, quantities four and six do not distinguish >= 5 from > 5: both versions reject four and accept six. Quantity five is the decisive input because the old predicate includes it and the new predicate excludes it. A test suite containing only ordinary values can therefore pass while the required threshold has changed.

Build a small truth table around each changed boundary. Include a value below, exactly at and above the threshold. For D2, the requirement accepts integers from zero through ten inclusive, and the proposal excludes zero. Values negative one, zero, one, ten and eleven test both ends of the interval. Only zero changes between the versions, but the other values help confirm which boundary moved and which remained the same. The point is not to generate a large table; it is to expose the specific contract affected by the edit.

A collection-level result can make the impact easier to see. D6 filters [4, 5, 6, 5]. The old version returns [5, 6, 5], while the proposal returns [6]. Both occurrences of five are lost. A reviewer who reports “one item changes” has confused one distinct threshold value with the number of affected records. Preserve the input order and duplicates when calculating expected outputs; the code does not deduplicate or sort.

State expected old and new outcomes before execution, then compare them with actual observations. This guards against accepting whatever the tool happens to print. If the observed result differs from your prediction, investigate whether the snippet, input or execution setup is wrong. A useful review records both the requirement expectation and the proposal's observed behavior. The requirement decides acceptability; the old version is a comparison point, not an unquestionable authority. An old version can also be wrong, so never define correctness as simply “unchanged from before.”

quantity	Old >= 5	New > 5	Meets inclusive requirement after change?
4	False	False	Yes for this input

quantity	Old >= 5	New > 5	Meets inclusive requirement after change?
5	True	False	No
6	True	True	Yes for this input

```
--- a/limit.py
+++ b/limit.py
@@ -1 +1 @@
-accepted = 0 <= quantity <= 10
+accepted = 0 < quantity <= 10
```

```
--- a/filter.py
+++ b/filter.py
@@ -1 +1 @@
-selected = [q for q in quantities if q >= 5]
+selected = [q for q in quantities if q > 5]
```

E05-X1 Produce exact boundary evidence

For D2, compute old and new acceptance for -1, 0, 1, 10 and 11. For D6, compute both selected lists for [4, 5, 6, 5]. Name the exact lost requirement in each proposal and recommend accept or revise.

Submit: A five-row truth table, two complete lists and two recommendations.

06 Review removal of a guard as a change in allowed states

A guard is a condition that prevents an operation in a prohibited state. C3's target.exists check protects an existing output in the stated fixture. Deleting the guard enlarges the set of states in which the write occurs. The happy path is still present, so a superficial demonstration using an absent target will not reveal the regression. The reviewer must test the state that the removed condition used to reject.

Use a sentinel file for a precise test. Before execution, create the target with the text KEEP followed by a newline. Set the proposed result to NEW followed by a newline. Run the old snippet in one isolated directory and the proposed snippet in another, each starting with the same sentinel. The old version should raise the stated error and leave KEEP unchanged. The proposed write_text call succeeds and replaces the sentinel with NEW. Record the target's content after each run, not just the exception or return code.

D4 expresses the same no-overwrite requirement through the file-open mode. The before version uses x and the proposal uses w. The change is one character, but it changes how an existing path is handled. With a fresh target both versions write NEW. With an existing target, x refuses creation while w truncates and writes. This contrast teaches a general review habit: identify the forbidden starting state and test the invariant that must survive it.

Do not overstate what C3's simple guard proves. A separate existence check followed by a write is not a complete concurrency-safety analysis. The exercise establishes a local

behavioral regression under controlled starting conditions, not a universal race-free implementation. A recommendation can say “restore the required no-overwrite behavior and verify an existing target remains unchanged” without asserting that the original snippet is the ideal production design. Requirements-based review is compatible with recognizing the limits of both versions.

```
--- a/write_mode.py
+++ b/write_mode.py
@@ -1,2 +1,2 @@
-with target.open("x", encoding="utf-8") as f:
+with target.open("w", encoding="utf-8") as f:
     f.write(result)
```

Starting target	Before mode x	After mode w
Absent	Creates NEW	Creates NEW
Contains KEEP	Raises FileExistsError; KEEP remains	Succeeds; content becomes NEW

E06-X1 Specify a preservation test

Write the full test plan and expected results for D4 using separate temporary directories. Include target contents before and after, expected failure category, fresh-target control case and recommendation. Do not run a test against a real report or a workspace repository.

Submit: A four-row before/after matrix and a concise review comment.

07 Distinguish cosmetic whitespace from meaningful whitespace

Whitespace is not one category of risk. In the foundation C1, a plain guide sentence changes “Use the kit.” to “Use the kit.” Under the supplied prose context, the words and instruction are unchanged, so formatting-only is a reasonable classification. That judgment depends on context. The same visual spacing change inside a string literal, a data format or indentation-sensitive code can affect exact output or control flow.

D1 changes a string label from Ready followed by two spaces and now to Ready followed by one space and now. The requirement says the exported label must have exactly one space. The proposal therefore fixes an observable output defect. Calling it formatting-only would miss that the string's value changes. The old string has ten characters and the new string has nine. Character counts can help demonstrate the difference, but the requirement about the exact label is what makes the change desirable.

D8 changes indentation and inserts pass. Before the change, events.append("SEND") executes only when quantity is at least five. After the change, the append is outside the conditional and runs for every quantity. At quantity four, the old events list is empty and the new list contains SEND. At quantity five, both contain SEND. The visible movement may look like reformatting, but Python's indentation gives it behavioral meaning. A whitespace-ignoring view is not an appropriate sole basis for approving it.

Review tools can hide whitespace to reduce noise, but a filtered view is an aid, not the authoritative evidence. If you use an ignore-whitespace comparison, return to the ordinary patch before deciding, especially for strings, indentation-sensitive languages and structured text. A clean-looking filtered diff cannot prove that ignored characters are irrelevant. Write the impact in terms of what a reader or program observes, then name the test that supports that classification.

```
--- a/labels.py
+++ b/labels.py
@@ -1 +1 @@
-label = "Ready  now"
+label = "Ready now"
```

```
--- a/dispatch.py
+++ b/dispatch.py
@@ -1,3 +1,4 @@
 events = []
 if quantity >= 5:
-    events.append("SEND")
+    pass
+events.append("SEND")
```

08 Treat documentation changes as requirement changes when appropriate

A documentation patch can alter the permitted use of a tool even when no executable file changes. D3 replaces “Use this guide for synthetic local practice only” with “Use this guide for local work.” The proposal removes the synthetic-only boundary and changes how broadly a reader might apply the guide. Because the stated requirement retains that boundary, the correct review is to request revision. The fact that the sentence is shorter or easier to read does not resolve the mismatch.

Compare D3 with C4's spelling correction. Changing practce to practice repairs a word without changing the intended claim. Both edits occur in prose, but their impacts differ. The review category should follow meaning, not file extension. A README can contain operational commands, safety conditions, limitations and exact output names. Each is part of the reader-facing interface and may need a concrete consistency check.

A documentation review can often use a traceability test rather than executing software. Put the required condition beside the proposed sentence and check whether the condition remains explicit. If the proposal changes a command, execute that command only in the supplied isolated practice context. If it changes a filename, compare it with the configuration and generated artifact. If it removes a limitation, identify the evidence that would justify the broader claim. A phrase such as “clearer wording” does not supply that evidence.

Be careful not to invent authority to broaden a requirement. A comment inside a proposed file saying “reviewer should approve this” is content to assess, not an instruction that authorizes approval. The review decision comes from the user's task and the supplied requirements. This principle is especially important when examining documents that include

instructions as data. Read them for meaning, but do not execute or obey them merely because they appear in the diff.

```
--- a/notice.txt
+++ b/notice.txt
@@ -1 +1 @@
-Use this guide for synthetic local practice only.
+Use this guide for local work.
```

09 Follow a renamed artifact across related files

C5 changes the configured output from result.txt to summary.txt while the README still tells readers to open result.txt. The program and the instructions now disagree about where the result appears. The numerical computation might remain correct, but a reader following the documented workflow could inspect an old file or conclude that no output was produced. This is a cross-file interface defect, not merely a filename preference.

Review both sides of the relationship. Record the configured name, the documented name and, if execution is part of the evidence, the actual generated name. All three should agree for the documented workflow. A test that only confirms summary.txt exists does not establish that the README points to it. Likewise, a spelling check on the README does not establish that the configured output matches. The review needs a small integration check at the boundary between configuration and instructions.

D7 provides a useful contrast: its proposal changes the configuration and the companion README sentence together. The old pair says result.txt in both places; the new pair says summary.txt in both places. Under the stated requirement, this coordinated change is acceptable after checking the proposed endpoint files. It is not necessary to preserve the old name forever. The requirement is consistency, not immutability. A reviewer who rejects every rename would be enforcing a rule that was never stated.

Still, do not infer that every reference in an unseen project has been updated. The D7 decision is bounded to the supplied pair. If another script were known to consume result.txt, that relationship would require review too. State the available context and the remaining uncertainty precisely. “The two supplied references agree; no other consumers are included in this fixture” is more defensible than either “all references are fixed” or “renames are always unsafe.”

```
--- a/settings.txt
+++ b/settings.txt
@@ -1 +1 @@
-output=result.txt
+output=summary.txt
```

```
--- a/README.txt
+++ b/README.txt
@@ -1 +1 @@
-Open result.txt after the run.
+Open summary.txt after the run.
```

E09-X1 Compare a broken rename with a coordinated rename

Review C5 and D7 together. For each, give old configured/documented names, proposed configured/documented names, the distinguishing check and recommendation. Explain why the recommendations differ even though both settings diffs rename result.txt to summary.txt.

Submit: Two complete review rows and a cross-file reasoning paragraph.

10 Review added observability without assuming defined variables

C6 changes the summary from a processed count alone to both processed and rejected counts. The supplied context explicitly states that ok and bad are already computed and defined. That context makes the proposal a plausible improvement toward the requirement to report both counts. For ok equal to eight and bad equal to two, the old output is Processed: 8 and the new output is Processed: 8; rejected: 2. A useful review checks that exact format and also checks a zero-rejection case so the absence of errors is visible rather than omitted.

D5 looks similar but changes the expression from bad to rejected while the only defined names remain ok and bad. The proposal raises NameError instead of producing a summary. A reviewer who sees the word rejected and assumes it must be the right variable has confused a label with an identifier. The literal label can say rejected while the value comes from a variable named bad. Meaningful names are helpful, but renaming a use without a corresponding definition breaks execution.

Check the context that determines whether the added expression is valid. Where does the value come from? Is it defined on every relevant path? Does it represent the required count rather than a different quantity? In this course the fixtures provide enough context to answer the immediate question. In an incomplete patch, request the missing definition or full function rather than inventing it. “Please show where rejected is assigned before this line” is a specific clarification that can resolve the uncertainty.

Observability is not the same as correctness of the underlying processing. A summary that prints eight processed and two rejected records tells the reader those recorded counts; it does not prove the classification of every record was correct. Keep the review recommendation aligned to the changed output requirement. The added summary is useful because it exposes available information, not because it guarantees that no hidden data problem remains.

```
--- a/summary.py
+++ b/summary.py
@@ -1 +1 @@
-print(f"Processed: {ok}; rejected: {bad}")
+print(f"Processed: {ok}; rejected: {rejected}")
```

Case	Defined values	Expected output or failure
C6 before	ok=8, bad=2	Processed: 8

Case	Defined values	Expected output or failure
C6 after	ok=8, bad=2	Processed: 8; rejected: 2
C6 after, zero rejected	ok=8, bad=0	Processed: 8; rejected: 0
D5 after	ok=8, bad=2 only	NameError for rejected

11 Use summary views without mistaking them for content review

A file list and a diff summary can help plan a review. They show which paths changed and roughly how much text was added or removed. They do not explain whether a condition was weakened, whether a string's exact value changed or whether a filename now contradicts a README. Treat these views as an index into the evidence. The decision still requires reading the relevant hunks and enough endpoint context to understand their effect.

Similarly, a whitespace check answers a limited question. git diff --check can report certain whitespace errors and conflict markers introduced by a patch. A clean result does not mean the patch meets the requirements. The threshold regression from >= to > can pass a whitespace check. The overwrite regression from x to w can also pass. Tool success must be interpreted in terms of what the tool actually checks, not transformed into a general approval signal.

A patch may also omit information that matters. Binary files do not necessarily have an informative line-by-line text diff. File mode changes, deleted files and renamed paths may appear in metadata rather than ordinary plus/minus lines. This course assesses text fixtures, so you are not expected to review an unseen binary format. The transferable habit is to mark the limitation and request a suitable inspection method rather than pretending every file can be judged from the same textual view.

Avoid using presentation cues as evidence. Green lines are additions in many interfaces, not endorsements. A small change count is not proof of low risk. A clean status after a commit means the working tree matches the recorded state, not that the recorded state is correct. Each signal has a specific meaning. Skilled review combines these signals with requirements and distinguishing tests while keeping their limits visible.

12 Run the synthetic review lab without touching a real repository

The supplied review_lab.py runs the numerical and file-preservation cases in temporary directories and prints structured observations. It also creates a new temporary Git repository to demonstrate the difference between staged and unstaged content. The repository has no remote, uses a fictional identity only in its local configuration and is removed when the temporary context ends. The script does not use a workspace repository, change global Git settings or perform network operations. Use an already installed Git and Python; software installation is outside this exercise.

The Git demonstration commits a seed guide, stages a spelling correction, then adds an unstaged note. It captures status, the ordinary diff and the staged diff. Read those three records together. The two comparisons should show different parts of the file's state. This lab creates changes only so you can inspect them in a disposable synthetic context; it does not authorize modifying another project or approving a real change.

The behavior cases test the exact authored snippets, not arbitrary instructions from external documents. They report old and new outcomes for boundary predicates, string values, summary names and file modes. Temporary targets use the sentinel KEEP and proposed content NEW. These records let you compare your predicted review evidence with actual execution. A negative case is informative when it reveals the expected violation or refusal, not only when the process exits successfully.

Save your actual lab output separately from the expected results in this chapter. Record the Git and Python versions used. If Git is unavailable, the program's Git section cannot establish the staged-versus-unstaged demonstration; report that limitation instead of substituting a screenshot from the text. You can still reason about the supplied patches and execute independent Python cases if the environment supports them. An honest partial evidence record is more useful than a full-looking record assembled from predictions.

```
python assets/review_lab.py
# The script prints JSON observations and uses temporary directories only.
# It does not modify the folder from which you invoke it.
```

E12-X1 Interpret staged and unstaged evidence

Run the supplied lab if the required tools are already available. Explain which comparison contains the staged spelling correction and which contains the later note. Give the expected short status for guide.txt and explain why an empty ordinary diff would not, in general, prove that nothing is staged. Separate your predicted result from actual execution evidence.

Submit: A three-state explanation, the two comparison meanings and actual or explicitly not-run observations.

13 Write review comments that specify a fix and a retest

A useful review comment contains a location, a consequence, a requirement and a next check. For C2, “The changed > comparison excludes quantity five, but the requirement includes quantities at least five. Retain inclusive behavior and retest four, five and six” is actionable. “This looks wrong” is not. The first comment gives the author enough information to understand and reproduce the issue without guessing what the reviewer meant.

Separate observed facts from suggestions. “With an existing target containing KEEP, the proposed write replaces it with NEW” is an observation if the test ran. “Use exclusive creation or another mechanism that preserves the no-overwrite requirement” is a possible remedy. The reviewer should not claim that one suggested implementation has been tested if it has not. In this course you can request restoration of the required behavior without rewriting the entire program.

When evidence is incomplete, make the clarification specific. For an added variable, ask where it is defined on the relevant path. For a renamed output, ask which supplied instructions and consumers must change with it. For a removed usage restriction, ask what approved requirement permits the broader claim. These questions target the uncertainty that affects the decision. A generic request for “more context” forces the author to guess which information matters.

Tone should remain factual and proportional. A one-character regression can be important without being evidence of carelessness or malicious intent. A typo correction can be accepted without a long ceremony. Group comments by the issue they resolve, but do not hide a critical deleted guard among minor wording suggestions. End the review with an explicit recommendation and the remaining acceptance checks. The reader should know whether the patch is ready, needs a known correction or awaits a specific fact.

14 Assemble a complete review ledger

A review ledger is a compact record of reasoning across several proposed changes. Use one row per change with columns for old meaning, proposed meaning, requirement impact, distinguishing check, evidence status and recommendation. This structure helps ensure that every supplied proposal receives a decision and that a critical issue is not lost among easy approvals. It also lets another reviewer challenge the reasoning without reconstructing your entire thought process from scattered comments.

For the six foundation changes, the ledger should distinguish the two harmless text corrections from the threshold regression, removed no-overwrite protection, inconsistent output name and useful added count. Do not collapse all documentation files into one category. C1 is formatting under the supplied context, C4 is a spelling correction, and C5 creates a workflow mismatch. The classification follows what changes for the reader or program, not the extension or number of lines.

Record whether each check was executed, predicted from source or still missing. A review can make a confident source-based statement about an obvious literal replacement while still needing a run to establish a particular filesystem observation. These evidence types are complementary, not competing. If a test was run, include the exact input and result. If it was not run, state the expected outcome and why it follows from the code, then avoid presenting it as an observation.

The final recommendation should identify blockers explicitly. C3 must not disappear into an aggregate score because five other rows look reasonable. A change set containing a known violation of the no-overwrite requirement is not ready merely because most edits are acceptable. At the same time, preserve the useful accepted parts in your ledger so the author can revise the problematic changes without losing track of the rest. Good review is selective and evidence-based, not blanket praise or blanket rejection.

E14-X1 Review the complete foundation change set

Produce a six-row review ledger for C1–C6. Include old/new meaning, requirement impact, distinguishing test and accept/revise/clarify recommendation. C3 must include exact existing-target preservation evidence. C6 must include both bad=0 and bad=2 expected outputs with ok=8. State an overall recommendation for the combined set.

Submit: A complete six-change ledger and an overall decision with blocking issues.

15 Check the whole changed region, not only the highlighted token

A changed token attracts attention, but its meaning depends on the surrounding region. D8 is a useful example because the visible comparison remains quantity >= 5. A reviewer who searches only for changed comparison operators might conclude that selection behavior is unchanged. The actual regression comes from moving events.append outside the conditional. The comparison still evaluates, but it no longer controls the append. Review the full control-flow relationship rather than assuming the most familiar token must be the important one.

Reconstruct execution at quantity four. In the old version, the condition is false, the indented append is skipped, and events remains empty. In the proposal, the condition is false, the pass statement is skipped, and the unindented append still runs. The result contains SEND. At quantity five, both versions append SEND, so that positive case alone cannot distinguish them. The negative case is necessary because the requirement forbids sending below the threshold.

The same whole-region habit applies to early exits. If a patch moves a write above a validation check, the validation may still exist but occur too late to prevent the side effect. If a patch changes a condition but leaves a contradictory explanatory comment, the program and documentation may diverge. These are not additional changes in the supplied D8 fixture; they are ways to read future patches with the same causal discipline. Ask which operation is controlled by which condition and in what order the effects occur.

Do not repair the code mentally while reviewing. It is tempting to read an unindented line as if it were still inside the block because that is where it should be. Instead, trace the proposal exactly as written and compare it with the requirement. A reviewer can state the intended fix afterward, but the observed proposal must remain the object of analysis. This discipline is especially valuable for small patches, where familiarity can encourage a quick scan instead of a precise reconstruction.

A concise review note for D8 should therefore name the moved append, the below-threshold input and the resulting list. “At quantity four, the proposal produces [SEND] because the append is outside the if block; the requirement permits SEND only at five or above” connects syntax to behavior. It is more informative than a generic style comment about indentation, and it gives the author a reproducible acceptance test after revision.

quantity	Before events	Proposed events	Requirement result
4	[]	["SEND"]	Proposal violates condition
5	["SEND"]	["SEND"]	Both satisfy this input

16 Recognize tests that give false reassurance

A test can pass and still be poorly chosen for the review question. Running the output writer only when the target is absent demonstrates ordinary creation, not preservation of an existing report. Testing the filter only at six demonstrates acceptance above the threshold, not inclusion of the exact threshold. Checking that the summary contains the word rejected does not prove that the referenced variable exists. Each of these tests observes something real but leaves the changed requirement untested.

Improve a weak test by naming the distinction it must expose. For an overwrite regression, vary the starting target state while keeping the result text constant. For a boundary regression, vary the input around equality. For a renamed artifact, compare the producer's name with the consumer instruction. For an undefined variable, supply only the names that the fixture actually defines. This approach produces a small, purposeful test set rather than a large collection of unrelated examples.

An error can also create false reassurance if it occurs for the wrong reason. Suppose a writer test fails because the parent directory does not exist. That failure does not show that an existing file was protected. The target-state setup must therefore be verified before execution. Likewise, a summary snippet that fails because of a transcription syntax error does not establish the intended NameError. Record the failure category and enough message detail to distinguish the intended case from a setup mistake.

The test harness itself should not silently repair the proposal. If a helper predefines rejected as an alias of bad, D5 will appear to work even though the supplied context does not define that name. If a helper deletes the target before every writer run, it prevents the preservation test from reaching the relevant state. Review the setup assumptions as carefully as the final

assertion. A good harness creates the required state and reports what happened; it does not make the proposal look correct by removing the condition under examination.

Finally, distinguish a comparison test from a requirement test. A comparison test shows that old and new differ. A requirement test says which outcome is acceptable. D1 differs because the new string has one space, and that difference is desirable. D2 differs because zero is excluded, and that difference is undesirable. Change alone is neither a defect nor an improvement. The requirement provides the direction of the judgment.

17 Trace evidence into a bounded recommendation

A review recommendation is an argument with a short chain of evidence. Start with the requirement, identify the changed expression or sentence, give the distinguishing input or comparison, state the resulting consequence, and then choose the recommendation. This chain makes the review auditable. Another person can check the requirement, reproduce the test or challenge a missing assumption without relying on the reviewer's confidence alone.

For D5, the chain is especially clear. The required summary uses the available processed and rejected counts. The context defines ok and bad only. The proposal references rejected. Executing with the supplied definitions raises NameError, so the summary is not produced. The recommendation is revise the unresolved identifier or provide a corresponding valid definition within the agreed scope, then rerun the exact output checks. Notice that the review does not need to speculate about how the mistake happened.

For D7, the chain supports acceptance rather than revision. The requirement is filename agreement. The proposed configuration names summary.txt, and the proposed companion instruction also names summary.txt. The supplied pair therefore agrees. Acceptance remains bounded to those references and the stated local task. If additional consumers are not supplied, do not claim they were checked. The review can be decisive about known evidence while transparent about its scope.

A useful recommendation also says what would change the decision. For C5, coordinating the README with the configured name and verifying the documented artifact would resolve the known mismatch. For C3, restoring no-overwrite behavior and proving the sentinel remains unchanged would resolve the blocking regression. For a genuinely missing variable definition, the needed evidence is the relevant source path, not a general promise that tests pass. These acceptance conditions help the author act without requiring another round of vague interpretation.

Avoid turning a recommendation into an external action. This course asks for a review sheet, not a submitted approval, merge or publication. The distinction matters because a well-supported technical opinion does not itself authorize changing a shared system. Keep the deliverable as a local record of findings and the next evidence needed. The quality of the work lies in the reasoning, not in clicking an approval button.

18 Apply the method to a mixed eight-change packet

The additional D1–D8 packet combines exact text, numerical boundaries, documentation scope, file preservation, identifier context, list behavior, cross-file naming and indentation. These cases are deliberately different so that one memorized classification cannot answer the whole packet. Use the same review structure for each, but let the subject matter determine the test. A string-length check is useful for D1, not for D4; a sentinel file is decisive for D4, not for D3.

Read the requirements in assets/review-cases.json before opening the combined patch. Each case includes the before and after text and a suggested test focus. The suggestions are prompts for evidence, not predetermined approvals. Compute your expected outcomes independently, then run the supplied lab if the tools are available. Compare actual observations with predictions, and record any discrepancy rather than editing the expected result to match an unexplained output.

The packet contains acceptable changes as well as regressions. Do not assume every exercise hides a defect. D1 repairs the exact label, and D7 coordinates the supplied filename references. The other cases need careful attention to their own requirements. A reviewer who rejects everything may avoid one kind of false approval but still fails to provide a useful decision. The objective is discrimination: identify which changes satisfy the contract, which violate it and which require missing evidence.

Your final ledger should be complete enough to stand alone. Include the concrete old and new outcomes, not merely labels such as behavior or documentation. For D6, give both full ordered lists. For D8, give both lists at the distinguishing negative input. For D4, give the preserved and replaced sentinel values. For D3, name the removed synthetic-only boundary. These details make the recommendation reproducible and prevent generic language from hiding an incomplete analysis.

Conclude with the accepted subset, the revision subset and any unexecuted checks. If every supplied fact is sufficient for a source-based decision, do not invent an ambiguity just to use the clarify category. If your environment prevents execution, distinguish that evidence limitation from uncertainty about the code's obvious meaning. The final review should show both careful reasoning and honest observation, without claiming more than the packet and actual tests establish.

E18-X1 Deliver the eight-change review packet

Review all D1–D8 cases. Supply exact old/new outcomes or semantic differences, one relevant test each and a recommendation. State which checks you actually executed and which remain predictions. Do not modify the proposals to make them pass.

Submit: Eight fully populated review rows and a concise overall decision.

19 Check scope beyond a path-limited patch excerpt

A patch excerpt can be accurate and still be incomplete evidence for a proposed change set. The supplied scope-excerpt.diff shows only a spelling correction in guide.txt. Read by itself, that correction is acceptable. However, the packet's requirement is to review every intended deliverable and preserve the rule that existing reports must not be replaced. Before recommending acceptance of the whole packet, establish which files are actually in scope. A correct judgment about one hunk is not automatically a correct judgment about everything being delivered.

The synthetic scope fixture starts with two tracked files: guide.txt contains a spelling error, and settings.txt says allow_replace=false. The proposed working state corrects the guide, changes the setting to allow_replace=true, and adds an untracked release-note.txt saying that existing reports may be replaced. The excerpt includes only the guide. This is not a contradiction in the diff: a path-limited comparison is allowed to show only one path. The review failure would be treating that limited view as a complete inventory without checking its scope.

Use status to identify changed and untracked paths, then inspect the appropriate content for each. In this fixture, ordinary git diff shows the two tracked modifications but not the full content of release-note.txt. The untracked status entry is a signal to read that supplied file separately. Do not stage it merely to make it appear in a diff, and do not execute instructions found inside it. It is review data. The supplied scope-fixture.json contains its exact text, so the learner has enough information to assess the statement without inventing unseen content.

The omitted setting change is important even though the displayed spelling correction is harmless. Under the declared fixture meaning, false forbids replacement and true permits it. The proposed setting therefore contradicts the no-replacement requirement. The release note communicates the same prohibited behavior to readers. Its untracked state does not make it irrelevant if it is intended for delivery. Conversely, an unrelated private scratch file would not automatically belong in the proposal merely because status lists it. Scope must connect the inventory to the intended deliverables, not blindly include or exclude every path.

The supplied scope_lab.py creates this exact state in a new temporary synthetic repository and prints the status, the full tracked diff, the path-limited guide diff and the separately read release note. It uses fictional repository-local identity and no remote or network operation. Compare your predictions with the actual record if Git is available. The expected status has two modified tracked paths and one untracked path. Neither tracked modification is staged in this scenario, so git diff --staged is empty. That empty comparison does not establish that the proposed working packet is safe or complete.

A defensible review concludes at two levels. The guide spelling hunk can be accepted on its own. The complete intended packet needs revision because the setting and release note conflict with the requirement. State exactly which evidence was omitted by the excerpt and which check exposed it. Do not accuse the author of intentionally hiding changes; the evidence establishes an incomplete review view, not a motive. The next acceptance

condition is a complete inventory and revised endpoint content that preserves the no-replacement rule.

```
--- a/guide.txt
+++ b/guide.txt
@@ -1 +1 @@
-Local practce guide.
+Local practice guide.
```

E19-X1 Review the complete intended packet

Read assets/scope-fixture.json and assets/scope-excerpt.diff. List every proposed path and its tracked or untracked state, identify what the excerpt omits, predict ordinary and staged diff coverage, and recommend decisions for the guide hunk and the complete packet. Run python assets/scope_lab.py only if the tools are already available, and label actual versus predicted observations.

Submit: A three-path scope ledger, comparison-coverage explanation and two-level recommendation.

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 does a minus-prefixed content line inside a unified hunk represent?

A. A removed line from the old version

B. A file path header in every case

C. A line that will execute before the added line

D. A test failure reported by Git

2. Why can changing >= to > matter despite changing one character?

A. Git treats every punctuation edit as a conflict

B. The line count determines the number of affected records

C. Equality at the boundary changes from included to excluded

D. The new code necessarily fails to parse

3. Which input distinguishes quantity >= 5 from quantity > 5?

A. 4

B. 6

C. Any negative integer

D. 5

4. What is the relevant consequence of C3 removing the existence guard?

A. All fresh-target writes stop working

B. An existing target may be replaced by write_text

C. The output becomes append-only

D. Only the exception wording changes

5. Why is C1 formatting-only under its supplied context?

A. Whitespace never matters in any text file

B. The diff has fewer than three lines

C. The same ordinary prose instruction remains after spacing cleanup

D. The file is not Python, so requirements cannot apply

6. What must be fixed before accepting C5 as a coherent workflow?

A. The configuration and README must agree on the output name

B. The threshold must be lowered to four

C. The old report must be deleted automatically

D. The summary must stop reporting rejections

7. Why can C6 be accepted with output checks?

A. Any added print call proves processing is correct

B. The new label defines bad automatically

C. Rejected records no longer exist after printing

D. It exposes the already defined rejected count required by the task

8. Which command inspects content staged for the next commit?

A. git diff only

B. git diff --staged

C. git status --short only

D. git log --oneline only

9. A required variable definition is not shown. What is the strongest review response?

A. Assume it exists because the name is descriptive

B. Approve because the patch is short

C. Name the missing definition and request the relevant source context

D. Reject every change in the repository

10. Which foundation requirement makes C3 a blocking issue?

A. Every file must keep its original length

B. Every guide must use identical spacing

C. Output names can never change

D. Existing output must not be overwritten

11. For D6 and input [4,5,6,5], what does the proposed >5 filter return?

A. [6]

B. [5,6,5]

C. [5,6]

D. [4,5,6,5]

12. Why is D1 more than formatting-only in its stated context?

A. Changing any string always breaks syntax

B. The exact exported label value changes to meet the one-space requirement

C. Only the source display changes; the exported value is unchanged

D. Collapsing a space inside a literal is equivalent to changing indentation

13. What does a successful fresh-target writer test fail to establish?

A. Whether the program can create a target at all

B. Whether the result text can be written

C. Whether the test used a new path

D. Whether an existing target remains unchanged

14. Why does D5 raise NameError under the supplied definitions?

A. The label rejected contains punctuation

B. ok is eight rather than zero

C. The proposal references rejected, but only ok and bad are defined

D. The old format string has too many fields

15. Which proposed pair satisfies D7's filename-agreement requirement?

A. Settings summary.txt and README summary.txt

B. Settings summary.txt and README result.txt

C. Settings result.txt and README summary.txt

D. Any pair if a report already exists

16. What does MM guide.txt mean in the ordinary short-status scenario taught here?

A. Two files have the same name

B. The file has both staged and later unstaged modifications

C. The file was merged twice

D. The file is untracked and ignored

17. Why can a clean git diff --check result not justify approval by itself?

A. It compares only the staged version even when run without --staged

B. It proves all changed branches were executed

C. A threshold or overwrite regression can contain no flagged whitespace issue

D. It proves exact output labels satisfy the specification

18. At quantity four, what distinguishes D8 after the append is moved outside the conditional?

A. Both versions produce [SEND]

B. Both versions produce []

C. The proposal cannot parse

D. Old events is []; proposed events is [SEND]

19. Which review comment is most actionable for C2?

A. The code feels different

B. Quantity five is now excluded despite the >=5 requirement; restore inclusive behavior and retest 4,5,6

C. Please improve everything

D. Approved because the patch is small

20. How should an unexecuted lab check be recorded?

A. As a prediction or not-run check, separate from actual observations

B. As passed if the model answer is known

C. As executed if the source was read

D. As irrelevant because review never needs evidence

Sources and further reading

Git: git diff
https://git-scm.com/docs/git-diff

Git diff format documentation
https://git-scm.com/docs/diff-format

Git status documentation
https://git-scm.com/docs/git-status