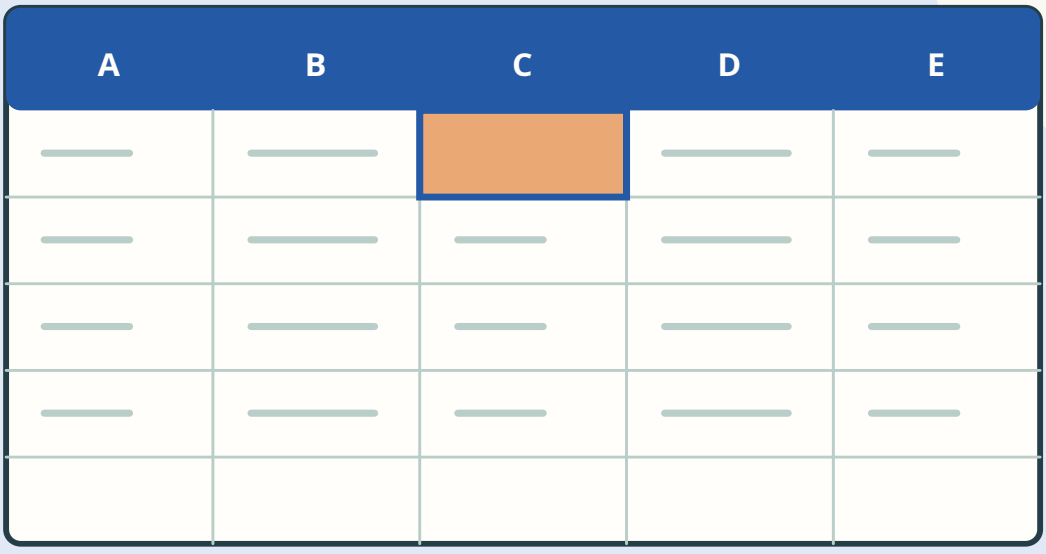
PYTHON CSV FILTERING

# Filtering CSV Rows with Python




ROWS / FILTER

DIGITAL & TECHNOLOGY

# Filtering CSV Rows with Python

## Learning outcomes

- Read CSV fields with a parser rather than splitting text on commas.
- Apply a numeric inclusive boundary and an exact status condition.
- Write a new output file preserving the header and source order.
- Distinguish logical CSV records from physical lines and inspect quoted commas, quotes and newlines.
- Trace exact-status and inclusive integer-threshold predicates without modifying original field strings.
- Preserve full selected rows, order and multiplicity across CSV serialization and reparsing.
- Enforce separate exclusive output creation and distinguish invalid input from valid empty results.
- Use independent decision ledgers and mutation tests to diagnose plausible filter defects.

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

## CSV is a format, not a simple comma split

A field can contain a comma inside quotes, so line.split(",") does not reliably reconstruct rows. Use csv.DictReader with newline="" and an explicit encoding. DictReader treats the first row as field names and gives subsequent values as strings. A quantity of "10" is still text until you convert it. Comparing numeric strings lexically can produce surprising order: "10" is less than "5" in a text comparison. Inspect the header before processing so a renamed or missing quantity column does not silently change the meaning of the operation.

## Separate parsing from selection

The filter here is status exactly equal to ready AND integer quantity greater than or equal to five. Both conditions must hold. The boundary includes five, but Ready with an uppercase R is not equal to ready under the stated rule. Do not silently normalize case unless the specification authorizes it. Parse quantity with int and handle a malformed value explicitly rather than guessing. The supplied practice data is valid, so a parse failure indicates a setup or copying problem. Keep the row as a dictionary after testing; this preserves its other fields, including quoted notes.

## Write only to a separate destination

Use csv.DictWriter with the original fieldnames, call writeheader(), and write the selected rows. Preserve input order unless sorting is requested. Compare resolved input and output paths before writing; a different-looking relative path can still point to the same file. Exclusive creation mode x can prevent accidental replacement of an existing output during practice. A new output with zero selected rows should still have its header, because “no matches” is a valid result. After writing, read the output with the CSV parser and reconcile selected ids, row count and field values against the source.

## Worked example

Worked example: records Q1,ready,5 and Q2,ready,4 produce only Q1 under quantity >= 5. A row Q3,Ready,9 is excluded because the status comparison is case-sensitive. The expression row["status"] == "ready" and int(row["quantity"]) >= 5 expresses both decisions. If the note is "pack, then check", DictReader returns that complete note as one field instead of splitting it into two.

## Foundation practice

Csv filename: requests.csv

### Csv text

id,status,quantity,note
R01,ready,4,below boundary
R02,ready,5,at boundary
R03,ready,6,above boundary
R04,hold,9,not ready
R05,Ready,8,case differs
R06,ready,10,"pack, then check"
R07,ready,0,zero
R08,hold,5,hold at boundary
R09,ready,12,large
R10,ready,5,second boundary

Condition: Exact status ready AND integer quantity >=5. No sorting or deduplication.

Output filename: selected_requests.csv

## Foundation tasks

- Predict selected ids and explain the two boundary rows.
- Implement a filter that checks distinct source/destination paths and preserves all four columns.
- Capstone: produce a new filtered CSV, reparse it, reconcile ids and quoted notes, and show the source remained unchanged.

# 01 The selection contract and the unchanged row

Filtering answers a narrow question: which existing records satisfy a stated condition? It does not mean repairing statuses, sorting priorities, summing quantities, or collapsing repeated identifiers. This course selects a row only when status is exactly ready and its quantity, interpreted as an integer, is at least five. The output preserves all four original fields and the relative source order of selected rows. These requirements define correctness together; a correct set of identifiers is not sufficient if a note is lost or the order changes.

The dispatch case describes synthetic equipment requests. It contains ordinary boundary quantities, mixed capitalization, a trailing space in status, a quoted comma, a quoted multiline note, an integer with leading zeros, and a repeated identifier. These are deliberate distinctions rather than dirty data to fix automatically. A row containing quantity 0005 is selected numerically, but the saved quantity field remains the original string 0005. Conversion is used to make a decision, not to rewrite the record.

Write the predicate in ordinary language before translating it into Python. Both conditions are required. The status comparison is exact. The threshold includes five. No other field influences selection. In particular, the identifier does not need to be unique, the note does not have to be nonempty, and the quantity is not used as an output sort key. If you find yourself inventing any of those rules, stop and compare your implementation with the stated contract.

The source remains a read-only input. A successful output is a newly created, separate CSV with the original header and only selected records. If no records match, the header still exists. If the input cannot be parsed under the declared assumptions, the worked implementation stops before creating an output. That failure policy is deliberately limited: this course does not teach repairing or quarantining individual invalid rows, which would require a different processing contract.

Use prediction, execution, and reconciliation as separate stages. First produce a hand decision ledger. Then run the supplied filter into a new destination or use the temporary-directory demonstration. Finally reparse the result and compare every field, including quoted notes and repeated rows. Keeping these stages distinct prevents the program's own output from becoming the only evidence that the program is correct.

| Requirement | Exact interpretation |
|---|---|
| Status | row["status"] == "ready" |
| Quantity | int(row["quantity"]) >= 5 |
| Combiner | Both conditions, not either condition |
| Preservation | Original four field strings, row multiplicity and source order |
| Destination | New separate CSV; header even with zero matches |

### E01-X1 Predict the dispatch selection

Inspect dispatch.csv with a CSV-aware reader or the supplied inspection program. Predict selected IDs in order, retaining repeated IDs. Explain D14, D16, D17 and D19.

Submit: An ordered ID sequence and four row-specific reasons.

# 02 CSV records are not physical text lines

A CSV record is structured by delimiter and quoting rules. A comma inside a quoted field belongs to that field rather than separating two columns. A newline inside a quoted field can also belong to the same logical record. These cases make line-by-line comma splitting unsuitable even when most rows look simple. The correct abstraction is a CSV parser that understands the format's state across delimiters, quotes, and physical lines.

In dispatch.csv, D15 has the note pack, then seal. The CSV writer surrounds that field with quotes so the embedded comma does not create a fifth column. DictReader returns the note as a single string without the syntactic outer quotes. In D20, the note contains an actual newline between first shelf and second shelf. That one logical record spans two physical lines in the source file. The selected output must preserve that internal newline as part of the note value.

The file's physical line count is therefore not its data-record count. The supplied inspect_csv.py prints a logical record number and reader.line_num, which reports the physical lines consumed by the parser. After the multiline D20 record, the physical count is one greater than a simple header-plus-record calculation would suggest. This distinction is useful for diagnostics but should not be confused with the filtering predicate.

Open CSV text files with newline="" and encoding="utf-8" so the CSV machinery can handle newline conventions itself while decoding is explicit. This setting is used on both reading and writing. Do not preprocess the file by splitting it into lines and stripping each line, because that can change a quoted field's whitespace or erase its internal line break before the parser sees it. Parse first, then inspect field values.

The default csv dialect used here is comma-delimited with double quotes and normal double-quote escaping. No dialect guessing is needed because the fixture format is declared. If a future source uses semicolons or tabs, that is a format contract to specify rather than a reason to try arbitrary delimiters until something looks tabular. A parser is reliable only when its configuration matches the source format.

```
"""Inspect logical CSV records without writing any file."""
from pathlib import Path
import argparse
import csv


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("source", nargs="?", type=Path,
                        default=Path(__file__).resolve().parent / "dispatch.csv")
    args = parser.parse_args()
    with args.source.open(encoding="utf-8", newline="") as handle:
```

```
        reader = csv.DictReader(handle, strict=True)
        print("header:", reader.fieldnames)
        for record, row in enumerate(reader, start=1):
            print(record, reader.line_num, repr(row), sep=" | ")


if __name__ == "__main__":
    main()
```

| Source feature | Parsed value | Meaning |
|---|---|---|
| D15 quoted comma | pack, then seal | One note field |
| D20 quoted newline | first shelf\nsecond shelf | One record with a two-line note |
| S31 doubled quotes | fragile "blue" kit | Quote characters are data |

### E02-X1 Count logical and physical positions

For dispatch.csv, record the logical record number and reader.line_num after D19, D20, the second D12, and D21. Explain why D20 advances the physical counter twice.

Submit: A four-row position ledger.

# 03 Headers establish the schema before selection

DictReader normally takes the first record as field names and uses them as keys for later records. That is convenient, but it does not prove that the source has the schema your program expects. A renamed quantity field, a duplicate header, or a different ordering can change what the dictionaries mean. The worked program checks that fieldnames equals the exact list id, status, quantity, note before reading data rows.

The exact header check is intentionally stricter than merely asking whether four expected names appear somewhere. It catches extra columns, missing columns, reordered columns, and duplicate names. For this exercise, the header is part of the output-preservation requirement, so there is no need to invent a remapping policy. A program designed for flexible schemas could define different rules, but those rules would need explicit tests and documentation.

A header-only file is valid under this contract. DictReader obtains the four field names and then yields no data rows. An entirely empty file is different: there is no schema to preserve, so fieldnames is None and the program raises ValueError. This is analogous to distinguishing an empty result table from an unknown table structure. Do not create a guessed header silently when the input contract says the original header must be preserved.

A row can have too many or too few fields even with a correct header. By default, DictReader stores extra values under a None key and supplies None for missing fields. The expanded worked program rejects these shapes before filtering. That check is not a repair procedure; it prevents a field-position error from being carried into an apparently successful output. The provided extra_field.csv and missing_field.csv are small negative fixtures for this boundary.

Duplicate identifiers in data are not duplicate headers. A repeated D12 record remains a valid separate request because uniqueness was not required. Be precise about the level at which a rule applies. Header uniqueness helps define field names; data-ID uniqueness would be an additional business rule. Confusing them can cause a filter to drop legitimate records while claiming to enforce schema quality.

```
FIELDS = ["id", "status", "quantity", "note"]
if reader.fieldnames != FIELDS:
    raise ValueError("Expected header: id,status,quantity,note")
if None in row or any(row[name] is None for name in FIELDS):
    raise ValueError("Wrong number of fields")
```

### E03-X1 Separate empty result from missing schema

Predict the behavior for header_only.csv, wrong_header.csv, extra_field.csv and missing_field.csv. State whether a new output is created by select when given a fresh destination.

Submit: Four input outcomes and destination-existence predictions.

## 04 Numeric comparison without destroying the original field

DictReader returns field values as strings in the normal mode used here. The quantity field "10" is not already an integer merely because its header says quantity. Comparing "10" >= "5" performs a lexical string comparison and gives False because the first character 1 sorts before 5. That result contradicts the intended numeric threshold. Convert the quantity to an integer for the decision.

The conversion and the preserved field have different roles. quantity = int(row["quantity"]) creates a numeric value for comparison. The original row dictionary remains unchanged. D16 therefore has numeric quantity five while its original field is "0005". D19 has numeric quantity six while its original field is "+6". The output writer receives the original row, not a reconstructed row with a normalized quantity string.

int accepts the controlled integer spellings used in these fixtures, including a sign, leading zeros, and surrounding whitespace. It rejects the decimal-point string "5.0" even though a person might regard it as the same mathematical amount as five. Do not introduce float conversion followed by int to make that input pass. Such a repair would also truncate nonintegral values and change the input contract. Here, invalid integer text causes a clear failure before output creation.

A negative integer is still a successfully parsed integer. D18's -2 fails the threshold because it is below five, not because decoding or integer parsing failed. Likewise, zero is valid numeric text but is excluded. Separate syntactic conversion failure from an ordinary False selection decision. This distinction lets a reviewer explain whether a record was not selected or whether the whole input could not be processed.

The code parses every row's quantity before applying the status-and-threshold condition. That creates a clear validation policy for the expanded worked program, even for held rows.

The original practice fixtures are all valid and retain their original outcomes. The purpose of this choice is not to add a new row-recovery system; it is to ensure that a successful report means all records in the examined input met the declared integer-text assumption.

| Original quantity string | int result | Threshold passes? | Saved field if selected |
|---|---|---|---|
| 4 | 4 | No | Not selected |
| 5 | 5 | Yes | 5 |
| 10 | 10 | Yes | 10 |
| 0005 | 5 | Yes | 0005 |
| +6 | 6 | Yes | +6 |
| 5 | 5 | Yes | 5 |
| 5.0 | ValueError | Not evaluated | No output created |

### E04-X1 Keep decision values separate from output values

For D16, D19 and S38, show original quantity text, parsed quantity, threshold result and exact output quantity text. For 5.0 state the declared failure.

Submit: A three-row value ledger and one error outcome.

# 05 AND, inclusive thresholds, and exact status

The predicate has two independent Boolean components. status_ok is row["status"] == "ready". quantity_ok is int(row["quantity"]) >= 5. Selection is status_ok and quantity_ok. A truth table makes the meaning visible: only True combined with True selects a row. An OR condition would admit both a held large request and a ready small request, which is not the stated rule.

The boundary symbol matters. Greater than or equal includes a ready row with quantity exactly five. Greater than excludes it. The dispatch fixture contains both a normal boundary row and a leading-zero representation of the same numeric boundary so you can distinguish boundary logic from string formatting. A held row at five remains excluded because passing one condition is insufficient.

Exact status also matters. Ready, READY, and ready followed by a space are not equal to ready. Trimming or lowercasing status may be useful in another workflow, but it is not permitted silently here. If the source contains a status that looks nearly correct, the report should reflect the declared comparison rather than guess the source author's intention. The fixture intentionally places these cases near ordinary rows to make unauthorized normalization detectable.

Parentheses can help a reader see logical grouping, although the simple two-part AND expression is already clear. Avoid replacing the status check with status != "hold". That alternative would admit unexpected statuses such as queued, Ready, or an empty string

whenever the quantity passes. A narrow positive condition is not generally equivalent to excluding one known negative value.

Before coding, fill a decision ledger with both component flags and the final decision. If your final selection differs from a colleague's, compare the first row where the flags diverge. A quantity flag disagreement points to parsing or the threshold; a status flag disagreement points to exact matching; equal flags with different final decisions point to the logical combiner. This is much more informative than debating the final count alone.

| status ok | quantity ok | AND selection | OR would select |
|---|---|---|---|
| False | False | False | False |
| False | True | False | True |
| True | False | False | True |
| True | True | True | True |

### E05-X1 Build a predicate decision ledger

For every dispatch record, give record number, ID, parsed quantity, status_ok, quantity_ok and selected. Keep the two D12 records separate.

Submit: A twelve-row ledger.

## 06 Stable selection preserves order and multiplicity

A filter traverses source records in order and retains a subsequence. If selected record A appeared before selected record B in the source, A remains before B in the output. This is stable selection. It does not require sorting because the source already provides the ordering to preserve. Adding a sort operation would be an unauthorized change even if it makes the result look tidier.

Multiplicity is equally important. The dispatch source contains D12 twice with different notes. Both records satisfy the predicate, so both appear in the output. Using a set of IDs loses multiplicity; using a dictionary keyed by ID may overwrite one record with the other. Either approach changes a row-filtering task into deduplication. A set can be useful for some comparisons, but it cannot establish that the required ordered sequence of rows was preserved.

The expected dispatch ID sequence is D12, D15, D16, D19, D20, D12. Its length is six even though it contains only five distinct ID strings. The two D12 notes identify distinct records: boundary kit and repeat request retained. If a result contains one D12 but the count still happens to be six because another wrong row was added, checking only the total would miss both errors.

Compare full dictionaries or ordered field-value lists when reconciling the output. An ordered list comparison detects missing rows, extra rows, reordered rows, changed fields, and lost duplicates. An unordered set comparison detects only some of those issues. Choose the

comparison according to the preservation requirement rather than convenience. The supplied verify_output.py uses list equality deliberately.

When presenting an evidence ledger, include source record numbers as well as IDs whenever IDs can repeat. A record number identifies the occurrence in the current input; an ID identifies the business label. They serve different purposes. Do not claim that the record number is a permanent external identity, because inserting an earlier source row would change it. For this controlled fixture, it is a precise reference to the observed sequence.

| Selected source record | ID | Note |
|---|---|---|
| 2 | D12 | boundary kit |
| 5 | D15 | pack, then seal |
| 6 | D16 | retain leading zeros |
| 9 | D19 | signed integer text |
| 10 | D20 | first shelf\nsecond shelf |
| 11 | D12 | repeat request retained |

### E06-X1 Detect a deduplication mistake

A colleague uses a dictionary keyed by ID and keeps the last occurrence of each selected ID, then emits those dictionary values in insertion order. Predict the resulting ID sequence and the surviving D12 note. Explain why the result fails.

Submit: Exact wrong sequence, note and row-count difference.

## 07 Destination safety before any output is opened

Writing output is a consequential step because the wrong path can destroy an input or replace earlier evidence. The worked function first compares source.resolve() with destination.resolve(). A spelling such as folder/../dispatch.csv can refer to the same path as dispatch.csv after resolution. Comparing the raw strings alone would miss that relationship. A same-path request is rejected before opening an output.

Exclusive creation mode x adds a different protection: it fails if the destination already exists. This prevents an earlier result from being silently replaced during practice. Mode w would truncate an existing file, while mode a would append another header or another batch of rows unless carefully managed. Neither behavior matches the requirement to create one new result file. Distinct path checking and exclusive creation are complementary safeguards.

Resolved path equality is not a complete filesystem identity proof in every environment. Hard links can give different paths to the same underlying file. In this program, an existing hard-link destination is still protected by exclusive creation, which refuses any existing target. Symbolic-link resolution can identify a same-source alias, and exclusive creation

prevents replacing an existing link. Do not claim that these simple checks solve every possible concurrent filesystem race or hostile path scenario.

Input parsing happens before output creation. If the header, field count, integer text, or CSV syntax is unacceptable, the function fails without producing a misleading partial result. This choice means selected rows are held in memory until validation completes. For these small practice datasets, that trade-off is deliberate. A production-scale transactional design would need additional choices about temporary outputs, atomic publication, and storage, beyond this course's narrow filtering task.

Use a destination in your own working output folder and choose a new filename for each run. The supplied demonstrate_filter.py avoids persistent outputs by using a fresh TemporaryDirectory for each case. It is safe to run repeatedly because each destination is new, and the delivered source files are read only. Do not delete an existing output merely to hide an exclusive-creation error; decide whether that file is evidence you need to retain.

```
if source.resolve() == destination.resolve():
    raise ValueError("Source and destination must differ")
# Read and validate the input before this point.
with destination.open("x", encoding="utf-8", newline="") as handle:
    writer = csv.DictWriter(handle, fieldnames=FIELDS)
    writer.writeheader()
```

### E07-X1 Predict protected write attempts

In a temporary test directory, consider a source file and three destinations: the same resolved source path, an existing unrelated file containing KEEP, and a fresh path. State the outcome and what happens to existing bytes.

Submit: Three outcomes with preservation consequences.

## 08 A complete filter with a clear success boundary

The complete filter_csv.py below is available as a learner asset. It uses two functions to separate reading and selection from destination creation. read_selected returns the number examined and a list of original selected dictionaries. select performs the path safety check, calls the reader, and writes a new CSV. This design makes the success boundary visible: no output is opened until the input pass completes.

The reader uses strict=True to request stricter CSV syntax handling. This is not a guarantee that every semantically incorrect dataset will be detected; the explicit header, row-width, and integer checks remain necessary. A parser knows quoting syntax, not the business meaning of ready or the required threshold. Each layer has its own responsibility, and a useful error message identifies which layer rejected the input.

The code raises ValueError for declared input-contract violations, rather than using assert as the public validation boundary. Assertions can be disabled with optimization and should not be the only protection against an unexpected header or source/output collision. The raised error carries a logical record number for malformed row content. That number is not a physical line number when a quoted field spans multiple lines.

The writer uses the original four field names in order and explicitly chooses LF as its record terminator. Parsed field values are preserved, but the output's raw quoting and line-ending style need not be byte-identical to the source. That is normal serialization. The source itself remains unchanged. Keep these two claims separate: preserving source bytes and preserving selected field values are different requirements.

The command-line interface requires a destination and accepts an optional --source path. Without --source, it reads dispatch.csv beside the script using __file__. This makes the packaged default independent of the terminal's working directory. Explicit relative arguments still use the caller's working directory. The program prints examined and selected counts only after the output has been written successfully.

```
"""Filter exact ready status and integer quantity >= 5 into a new CSV."""
from pathlib import Path
import argparse
import csv

FIELDS = ["id", "status", "quantity", "note"]


def read_selected(source):
    selected = []
    examined = 0
    with Path(source).open("r", encoding="utf-8", newline="") as handle:
        reader = csv.DictReader(handle, strict=True)
        if reader.fieldnames != FIELDS:
            raise ValueError("Expected header: id,status,quantity,note")
        for record, row in enumerate(reader, start=1):
            examined += 1
            if None in row or any(row[name] is None for name in FIELDS):
                raise ValueError(f"Record {record}: wrong number of fields")
            try:
                quantity = int(row["quantity"])
            except ValueError as error:
                raise ValueError(
                    f"Record {record}: quantity is not integer text"
                ) from error
            if row["status"] == "ready" and quantity >= 5:
                selected.append(row)
    return examined, selected


def select(source, destination):
    source = Path(source)
    destination = Path(destination)
    if source.resolve() == destination.resolve():
        raise ValueError("Source and destination must differ")
    examined, selected = read_selected(source)
    with destination.open("x", encoding="utf-8", newline="") as handle:
        writer = csv.DictWriter(handle, fieldnames=FIELDS,
                                lineterminator="\n")
        writer.writeheader()
        writer.writerows(selected)
    return {"examined": examined, "selected": len(selected)}

def main():
```

```
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("destination", type=Path)
    parser.add_argument("--source", type=Path,
                        default=Path(__file__).resolve().parent / "dispatch.csv")
    args = parser.parse_args()
    result = select(args.source, args.destination)
    print(f"examined={result['examined']},selected={result['selected']}")


if __name__ == "__main__":
    main()
```

# 09 Writing valid CSV rather than manually joining strings

A selected dictionary is not yet a serialized CSV record. DictWriter knows the field order and applies quoting when a field contains a comma, quote, or newline. Manually joining values with commas would lose that protection. In the dispatch output, the D15 note needs quotes around the field. In the studio output, the internal quote characters around blue are doubled inside the quoted field according to the chosen CSV format.

Call writeheader once before writing data rows. The header is required even when the selected list is empty. This allows a downstream reader to identify the columns without guessing. Writing a header for every row would create fake data records in the middle of the file. Omitting it would make DictReader treat the first selected request as field names unless the caller supplied an external schema, which is not this workflow.

Keep the original dictionary when writing. A reconstruction that includes only id and quantity loses status and note. A reconstruction using int(quantity) changes leading zeros and surrounding spaces. A reconstruction that joins a multiline note with spaces changes its content. These modifications might be appropriate under a separate transformation request, but a row filter is expected to preserve the selected row's values.

Exact output bytes are predictable for the supplied implementation because it fixes field order and LF record terminators. However, two valid CSV serializations can represent the same records with different unnecessary quoting or different supported record terminators. The main semantic comparison should reparse the output and compare fields, order, and multiplicity. If a task also requires a particular serialization style, test that separately and state it explicitly.

A quoted field that begins with a formula-like character remains data in this course; no spreadsheet application is opened or executed. This material does not claim that arbitrary CSV content is safe to open in every external application. The authorized operation is local parsing and writing of synthetic strings. Preserve the boundary between creating a text data file and executing or interpreting its contents in another tool.

```
id,status,quantity,note
D12,ready,5,boundary kit
D15,ready,10,"pack, then seal"
D16,ready,0005,retain leading zeros
D19,ready,+6,signed integer text
```

```
D20,ready,12,"first shelf
second shelf"
D12,ready,5,repeat request retained
```

### E09-X1 Specify the complete dispatch output

Write the exact CSV serialization produced for dispatch.csv by the supplied writer, including its header, repeated record, preserved quantity strings and multiline field. Use LF record terminators.

Submit: Complete output text, not only selected IDs.

# 10 Reparsing the result as an independent evidence step

A successful write call proves that data was sent to the destination handle, not that the resulting rows match the specification. Reopen the output with the CSV parser and examine its header and records. This catches missing headers, broken manual quoting, omitted fields, changed values, lost duplicates, and unexpected order. A file's existence or nonzero size is not sufficient evidence of a correct filtered result.

The supplied verify_output.py recomputes expected rows from the source and compares the entire ordered output list. It also checks the output header. This is a useful automated consistency check, but it shares the selection function with the writer. A bug in that common predicate could make both agree on the same wrong selection. Therefore the hand decision ledger and boundary examples remain necessary independent evidence.

For dispatch.csv, compare the full sequence of six selected dictionaries, not just a count of six. Check that D15's comma note is a single value, D20's note contains an internal newline, D16 retains 0005, and both D12 occurrences remain in their original positions. For studio.csv, check doubled-quote decoding in S31 and the quantity spaces in S38. Each field-level check targets a plausible serialization or preservation mistake.

A useful reconciliation record has three layers: expected source occurrences, observed parsed output records, and a pass/fail explanation for each preservation rule. A row count may agree while content differs; distinct IDs may agree while multiplicity differs; all field values may agree as a set while order differs. State which comparison you performed rather than using the single word verified for every kind of evidence.

The verifier is read only. It does not fix an incorrect output or rewrite the source. If it raises ValueError, keep the failed output long enough to inspect the mismatch, then create a new destination for a corrected run. Do not automatically overwrite the evidence that explained the defect. The separation between producing a result and inspecting it helps you diagnose the actual cause.

```
"""Reparse a candidate output and compare every field with expected rows."""
from pathlib import Path
import argparse
import csv
from filter_csv import FIELDS, read_selected

def verify(source, destination):
```

```
    if Path(source).resolve() == Path(destination).resolve():
        raise ValueError("Source and destination must differ")
    examined, expected = read_selected(source)
    with Path(destination).open(encoding="utf-8", newline="") as handle:
        reader = csv.DictReader(handle, strict=True)
        if reader.fieldnames != FIELDS:
            raise ValueError("Output header mismatch")
        actual = list(reader)
    if actual != expected:
        raise ValueError("Output rows differ in content, multiplicity, or order")
    return {"examined": examined, "selected": len(actual)}


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("source", type=Path)
    parser.add_argument("destination", type=Path)
    args = parser.parse_args()
    result = verify(args.source, args.destination)
    print(f"verified={result['selected']},examined={result['examined']}")


if __name__ == "__main__":
    main()
```

### E10-X1 Choose comparisons that catch four defects

Consider four candidate outputs: one drops the last D12; one swaps D15 and D16; one changes 0005 to 5; one removes the comma from pack, then seal. State whether row count, ordered IDs, and full ordered-row comparison detect each defect.

Submit: A four-by-three diagnostic matrix.

# 11 Zero matches, header-only inputs, and honest counts

A filter can succeed while selecting nothing. no_matches.csv has three valid records: a held large request, a ready request below the threshold, and a capitalized status. None meets both conditions. The output contains the four-field header and no data rows. examined=3 and selected=0 is a successful result, not an exception or a reason to invent a placeholder request.

header_only.csv is a different valid case. Its schema is present, but there are no data records to examine. It produces the same header-only output with examined=0 and selected=0. The output files have the same data content, but the processing summaries communicate different source situations. This is why retaining an examined count can be useful even when the selected rows are identical.

An entirely empty source is not one of these valid cases because it lacks the declared header. The reader rejects it. Likewise, an invalid quantity does not merely mean that row is a nonmatch. It means the expanded worked program cannot complete its stated input pass, so no output is created. Distinguish a successful empty selection from a rejected input and from an inaccessible file.

The equation selected + excluded = examined holds for a successful pass over valid records. For dispatch.csv, six selected and six excluded account for twelve examined records. For studio.csv, five selected and four excluded account for nine. These reconciliations are useful totals, but they still do not prove which rows were selected. Combine them with the decision ledger rather than treating the equation as a substitute for it.

Avoid labels such as processed when their meaning is unclear. Does processed count logical records, physical lines, selected rows, or all rows attempted before failure? The supplied program prints examined only after a complete successful pass, so its meaning is stable. On failure it raises an error rather than printing a partial examined summary that could be mistaken for a complete result.

| Input | Examined | Selected | Excluded | Successful output |
|---|---|---|---|---|
| dispatch.csv | 12 | 6 | 6 | Header plus six records |
| studio.csv | 9 | 5 | 4 | Header plus five records |
| no_matches.csv | 3 | 0 | 3 | Header only |
| header_only.csv | 0 | 0 | 0 | Header only |

### E11-X1 Reconcile empty and nonempty cases

Run demonstrate_filter.py. Record all four summary lines exactly, including ordered IDs and unchanged=True. Explain why the two zero-selected cases have different examined counts.

Submit: Four console lines and a source-volume explanation.

## 12 Controlled mutations that expose a wrong predicate

A useful test changes one meaningful property at a time in an independent temporary copy. If a ready quantity changes from four to five, the selected count should increase by one. If a selected status changes from ready to hold, it should decrease by one. If only an unselected note changes, selection should remain identical. These predictions connect the test to the contract rather than merely asking whether the program runs.

For dispatch.csv, changing D11's quantity from 4 to 5 adds D11 at the beginning of the selected sequence. Changing D15's status to hold removes that one selected record. Changing D14's status from Ready to ready adds D14 before D15. Trimming D17's trailing status space would also add a row, but doing so in the real filter would be an unauthorized normalization. A controlled mutation is a test input change, not permission to change the delivered source.

Test incorrect implementations as well as correct input changes. An OR predicate selects more dispatch records than the correct AND predicate. A strict > 5 threshold loses the three selected occurrences whose numeric quantity is five. Lexical comparison loses "10", "0005",

"+6" and "12" relative to the string "5" in this particular fixture. Each wrong rule has a concrete, explainable signature.

A mutation test is meaningful only if the expected result is established independently. Do not run the modified program first and then label its output expected. Predict the exact selected occurrence sequence from the changed fixture, run the program on a temporary copy, and compare. Keep the original fixture bytes unchanged before and after the experiment so the test does not contaminate later evidence.

The test suite can use assert internally because it is checking known expectations in a controlled verification run. The learner-facing filter still uses explicit exceptions for invalid input and unsafe destinations. Keep that distinction clear: assertions in a test are not a substitute for runtime validation in a program that accepts caller-supplied paths and data.

**E12-X1 Predict three independent dispatch mutations**

In separate temporary copies: A changes D11 quantity to 5; B changes D15 status to hold; C changes D14 status to ready. Give each ordered selected ID sequence, retaining the two D12 occurrences.

Submit: Three exact sequences.

# 13 The studio case: preserve quotes, spaces, and Unicode

The independent studio.csv case uses a different row arrangement and different content from the dispatch demonstration. It contains nine records and five selected occurrences. The first selected note contains literal quote characters, a later note contains several commas, another includes café, one quantity includes surrounding spaces, and the final note contains an internal newline. These features test field preservation without changing the core selection rule.

S31 is selected with quantity eight and note fragile "blue" kit. When serialized, the quote characters are represented by doubled quotes inside a quoted field. When reparsed, the note must again contain the original quote characters, not doubled literal quotes. This round trip illustrates the difference between CSV syntax and the field value represented by that syntax.

S38 has quantity text consisting of a space, 5, and another space. int interprets it as five, so the threshold passes. The exact original string remains in the output. The writer may not need to quote that field under its default minimal quoting rules, but the spaces are still part of the value. Do not strip every output field merely because numeric conversion tolerated edge whitespace.

The accented character in café is preserved through explicit UTF-8 reading and writing. This is not a claim that every source encoding is supported. The fixture's declared encoding is UTF-8, and a different encoding would need a separate choice. The CSV parser solves delimiter and quoting structure; it does not automatically detect or repair text encoding.

Prepare the studio result without copying the dispatch answer and replacing IDs. The selected sequence is determined by its own records, and the note fields have different

preservation risks. A strong independent case should force the learner to reapply the rule, inspect new boundaries, and compare exact values. Reusing the same row pattern with renamed labels would provide much weaker evidence of understanding.

| id | status | quantity | note |
|---|---|---|---|
| S31 | ready | 8 | fragile "blue" kit |
| S32 | ready | 3 | below |
| S33 | hold | 20 | hold |
| S34 | ready | 5 | take, inspect, pack |
| S35 | READY | 7 | upper case |
| S36 | ready | 11 | café display |
| S37 | ready | 0 | zero |
| S38 | ready | 5 | retain numeric spaces |
| S39 | ready | 6 | line one<br>line two |

### E13-X1 Deliver the independent studio result

Produce the complete filtered studio CSV in a new temporary or learner-owned destination. Reparse it and give every selected field in order, plus examined/selected/excluded counts and source-byte evidence.

Submit: Five complete rows, 9/5/4 reconciliation and unchanged source evidence.

# 14 Debugging by layer rather than by changing the data

When a result is wrong, identify the first layer that disagrees with the contract. If the header is unexpected, inspect schema and delimiter configuration. If a note appears split into multiple fields, inspect quoting and parser use. If "10" is excluded while "6" is included, inspect numeric conversion. If Ready is selected, inspect unauthorized case normalization. If the right rows appear in the wrong order, inspect sorting or unordered containers.

A failure to create an output has a different diagnostic path. Same resolved source and destination is a deliberate ValueError. An existing destination produces FileExistsError through exclusive creation. A missing destination parent can produce an operating-system error. Invalid input causes a parsing or validation failure before the output is opened. Do not respond to all of these by switching to mode w, because that removes a protection without addressing the actual cause.

Reproducible diagnostics should identify the relevant source and condition without assuming exact machine-specific error wording. For a wrong header, record the observed field names

and the expected list. For invalid integer text, record the logical record number and the field value in a representation that exposes spaces. For wrong output content, show the first differing ordered row and field. Those observations make a correction reviewable.

Keep the original failed artifact when it helps explain the defect. If a manually joined output split the D15 note, reparse that candidate and show the extra field before replacing it with a new file. The evidence establishes why a CSV writer was necessary. Simply deleting the wrong output and presenting a later correct one loses the opportunity to connect the symptom to the cause.

Do not expand the task into general data cleaning during debugging. The correct response to an exact-status mismatch is to explain the mismatch, not silently strip and lowercase the source. The correct response to 5.0 is the declared failure, not a guessed conversion. Debugging repairs the implementation against the contract; it does not rewrite the contract to excuse the implementation.

**E14-X1 Identify four concrete defects**

For the dispatch fixture, give selected ID sequences produced by OR instead of AND, > instead of >=, lexical quantity comparison instead of int, and status.strip().lower() instead of exact status. Explain one revealing row for each.

Submit: Four wrong sequences and four causal diagnoses.

# 15 Two serializations can preserve the same records

A preservation claim must say what is preserved. The source file's bytes should remain exactly unchanged. The selected output must preserve field values, order, and multiplicity. These requirements do not imply that the output uses exactly the same optional quotation style as the source. A CSV writer may quote every field or only fields that require quoting, and both forms can decode to the same records.

The supplied compare_serializations.py demonstrates this distinction entirely in memory. It serializes a header and one C1 request twice, once with minimal quoting and once with all fields quoted. Both forms preserve the quantity string 0005 and the note pack, then seal. The literal text differs because the all-quoted version adds quotes around fields that do not otherwise need them. Reparse both with csv.reader and the resulting nested lists are equal.

This is not permission to ignore all textual differences. Removing a comma from inside the note changes a field value. Changing 0005 to 5 changes the stored quantity string even though both parse numerically to five. Replacing an embedded newline with a space changes the note. A semantic comparison identifies these differences after parsing, whereas a comparison of selected counts cannot. The key is to choose the right level for each requirement.

When a reviewer asks for exact output text, use the supplied writer's declared LF terminator and minimal quoting to make the expected bytes unambiguous. When a reviewer asks whether an independently produced CSV contains the correct selected rows, reparse it and compare the ordered fields. If the external contract requires a particular dialect, include that

contract as an additional check rather than assuming every syntactically valid CSV is interchangeable for the recipient.

The in-memory experiment also demonstrates why a representation is useful when showing serialized text in a report. repr exposes newline escapes and quote characters without confusing a field's internal newline with a new display paragraph. However, the representation itself is not the CSV file. Do not save the surrounding Python representation quotes or literal backslash escapes as though they were the actual serialized bytes. The complete script provides a runnable way to observe both layers without editing any delivered fixture.

```
"""Compare two in-memory CSV serializations of the same field values."""
import csv
import io


def serialize(quoting):
    buffer = io.StringIO(newline="")
    writer = csv.writer(buffer, quoting=quoting, lineterminator="\n")
    writer.writerow(["id", "status", "quantity", "note"])
    writer.writerow(["C1", "ready", "0005", "pack, then seal"])
    return buffer.getvalue()


def main():
    minimal = serialize(csv.QUOTE_MINIMAL)
    quoted = serialize(csv.QUOTE_ALL)
    first = list(csv.reader(io.StringIO(minimal, newline="")))
    second = list(csv.reader(io.StringIO(quoted, newline="")))
    print(f"same_rows={first == second},same_text={minimal == quoted}")
    print("minimal:", repr(minimal))
    print("all_quoted:", repr(quoted))


if __name__ == "__main__":
    main()
```

### E15-X1 Compare values and serialization

Run compare_serializations.py. Give its first output line and both serialized strings. Explain why 0005 remains important even though the selection decision uses integer five.

Submit: Two exact strings, comparison flags and a field-preservation explanation.

# 16 Capstone: publish a new result with bounded evidence

The final deliverable should let another reader reproduce the decision without trusting your conclusion. Include the source identity, exact header, predicate, ordered selected rows, new destination identity, and reparse comparison. Include a byte comparison showing that the source observed before and after processing was unchanged. If you used the temporary demonstration, state that its output files existed only inside temporary directories and were removed when those contexts ended.

For the foundation requests.csv, retain the original expected sequence R02, R03, R06, R09, R10. The note pack, then check remains a single field. The additional dispatch and studio cases extend the evidence but do not replace this regression check. A change that handles multiline notes but breaks the original inclusive boundary is not an improvement under the course contract.

Use the complete demonstrate_filter.py when you want a repeatable run that does not leave output files beside the sources. It creates a fresh destination for each case, runs the filter, reparses the output, checks source bytes, and prints a compact summary. Because it imports the same predicate, your independent hand ledgers remain part of the assessment. Automation is most useful when you understand which assumptions are shared and which evidence is independent.

The final review should also include negative cases. Show that an existing destination remains unchanged, a source-alias destination is rejected, an unexpected header does not create a partial output, and a decimal-point quantity fails the integer-text contract. These are not optional polish: they demonstrate that the safe success boundary is enforced when the task cannot be completed as specified.

Keep claims proportionate to the exercise. You have demonstrated filtering of declared UTF-8 comma-delimited files into new local outputs with preserved field values, order, and multiplicity. You have not tested every CSV dialect, every filesystem race, or spreadsheet application behavior. A precise report makes the result more useful because it tells the next person exactly what they can rely on and what would require a new test.

```
"""Write only within a temporary directory; preserve supplied sources."""
from pathlib import Path
from tempfile import TemporaryDirectory
import csv
from filter_csv import select
from verify_output import verify


def main():
    base = Path(__file__).resolve().parent
    for name in ("dispatch.csv", "studio.csv", "no_matches.csv",
                 "header_only.csv"):
        source = base / name
        before = source.read_bytes()
        with TemporaryDirectory() as temporary:
            destination = Path(temporary) / "selected.csv"
            result = select(source, destination)
            verify(source, destination)
            with destination.open(encoding="utf-8", newline="") as handle:
                rows = list(csv.DictReader(handle))
            if source.read_bytes() != before:
                raise RuntimeError("Source bytes changed")
            ids = "|".join(row["id"] for row in rows) or "(none)"
            print(name, result["examined"], result["selected"], ids,
                  "unchanged=True", sep=" ; ")


if __name__ == "__main__":
    main()
```

### E16-X1 Submit the complete filtering evidence

For dispatch.csv and studio.csv, provide complete parsed selected rows, counts, unchanged-source evidence and successful reparse verification. Include no_matches.csv and header_only.csv results, plus same-source, existing-output, wrong-header and invalid-quantity negative outcomes.

Submit: Two exact row tables, four case summaries and four protected-failure records.

# 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. Why is DictReader preferable to splitting each physical line on commas?**

A. It automatically converts quantity fields to integers

B. It guarantees every header matches the required schema

C. It removes repeated IDs before filtering

D. It recognizes quoted delimiters and multiline field structure

**2. What is the normal DictReader value for a quantity field containing 10?**

A. Integer 10 because the header is quantity

B. String "10" until explicitly converted

C. Float 10.0 because the field is unquoted

D. Integer 10 only if all rows are numeric

**3. Which predicate implements the complete selection rule when quantity is already an integer?**

A. status == "ready" and quantity >= 5

B. status == "ready" or quantity >= 5

C. status != "hold" and quantity >= 5

D. status == "ready" and quantity > 5

**4. A row has status ready and integer quantity five. Is it selected?**

A. Yes, because the threshold includes five

B. No, because the quantity must exceed five

C. Only if the ID has not appeared earlier

D. Only if the note contains no comma

**5. How does the exact status rule treat Ready?**

A. It matches because Python string equality ignores case

B. It does not match ready

C. It matches only when quantity is ten or more

D. It matches after the CSV parser standardizes capitalization

**6. What should a successful zero-match output contain?**

A. The first excluded row as an example

B. No bytes at all

C. The original four-field header and no data rows

D. A row containing selected=0 instead of the header

**7. Why compare resolved source and destination paths?**

A. Resolution converts quoted fields safely

B. Resolution guarantees there are no hard links anywhere

C. Different path spellings can identify the same location

D. Resolution selects only existing files

**8. Which output mode refuses an already existing destination?**

A. w

B. a

C. r+

D. x

**9. How is the quoted source note "pack, then seal" represented after parsing?**

A. Two fields, pack and then seal

B. One string containing the comma

C. One string with the comma removed

D. One string retaining the outer CSV syntax quotes

**10. Which ordering is required for the selected output?**

A. The relative order of selected source occurrences

B. Ascending numeric quantity

C. One occurrence per ID in alphabetical order

D. All boundary quantities before larger quantities

**11. D16 has quantity text 0005 and status ready. What should the output quantity field be?**

A. 5 because int returns five

B. 0005 because conversion is only for the decision

C. 5.0 because CSV stores numeric fields as decimals

D. The row should be rejected because of leading zeros

**12. Why does D20 occupy one logical record but two physical source lines?**

A. Its ID occurs twice

B. Its quantity is larger than five

C. Its quoted note contains an embedded newline

D. DictReader inserts an extra blank line

**13. Which comparison detects a changed note when row count and ID order are unchanged?**

A. Full ordered-row field equality

B. Selected row count only

C. Set equality of IDs

D. Maximum selected quantity

**14. The dispatch source has two selected D12 occurrences. What should the filter do?**

A. Keep the first because it appeared earlier

B. Keep the last because it is newer in the file

C. Combine their quantities into one D12

D. Retain both in their source positions

**15. What happens to a fresh destination when invalid_quantity.csv contains 5.0?**

A. It contains the first valid row and no header

B. It is not created because validation fails before writing

C. It contains both rows after float conversion

D. It is created as a successful header-only result

**16. Which statement distinguishes header_only.csv from no_matches.csv?**

A. Only no_matches has a valid schema

B. Only header_only can produce a header-only output

C. Both select zero, but examined is zero versus three

D. They must have different output headers

**17. A destination already exists with the bytes KEEP. What does the supplied select function do after a valid input pass?**

A. Appends the new CSV after KEEP

B. Replaces KEEP because the path differs from the source

C. Deletes the destination and retries

D. Raises FileExistsError and leaves KEEP unchanged

**18. What is the strongest limitation of verify_output using the same read_selected function as the writer?**

A. A shared predicate bug can make both agree incorrectly

B. It cannot detect output row reordering

C. It cannot inspect quoted notes

D. It treats valid alternate quoting as different parsed record values

**19. Two CSV texts differ only because one quotes every field. Reparsed ordered fields are equal. What follows?**

A. The source file must have been modified

B. One serialization is necessarily invalid

C. They preserve the same record values despite different text

D. Their numeric fields must now be integers

**20. Which input would the exact header check reject?**

A. id,status,quantity,note followed by no records

B. id,state,quantity,note followed by otherwise valid records

C. A valid row with repeated ID D12

D. A valid note containing a quoted comma

## Sources and further reading

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

Python built-in functions
https://docs.python.org/3/builtins/functions.html

Python pathlib
https://docs.python.org/3/library/pathlib.html

Python Errors and Exceptions
https://docs.python.org/3/tutorial/errors.html