Module 14 of 20

Files and data formats

open with modes, encodings and with; pathlib for building, querying and listing paths; CSV with DictReader/DictWriter; JSON with custom encoders and dataclass round trips; bytes with struct, byte order, base64 and hashlib; and sqlite3 with parameters, transactions and aggregation.

Programs live on data that arrives in files and formats, and Python's standard library handles the formats that matter without a third-party package: text files with an explicit encoding, paths as objects, CSV with its quoting rules, JSON with hooks for the types it does not know, fixed binary layouts with `struct`, and a real SQL database in `sqlite3`. This module teaches each on the terms that keep data safe — state the encoding, close the file, quote the field, parameterise the query, order the output — and uses the judge's writable working directory so that every exercise creates, reads and removes real files.

Reading and writing files covers `open`'s modes and encoding, `with`, line iteration versus `read`, `write` versus `print(file=)`, safe replacement and `io.StringIO`. pathlib covers joining with `/`, the parts, queries, `read_text` / `write_text`, `iterdir` / `glob` / `rglob` (sorted), creating and deleting, and pure paths. CSV covers `reader` / `writer`, `DictReader` / `DictWriter`, `newline=""`, delimiters and quoting, and reading from stdin and strings. JSON in depth covers `default=`, `asdict` round trips, `object_hook`, `parse_float=Decimal`, validation, JSON Lines, `tomllib` and `configparser`. Bytes covers `bytes` / `bytearray`, byte order, `struct`, binary files, `base64` and `hashlib`. sqlite3 covers connecting, parameters, queries, `Row`, transactions with `with con:`, `GROUP BY` and joins.

The exercises are whole programs: a file written and read back with its byte size, an append-only log, path decomposition with pure paths, a directory tree built and globbed, CSV totals by category, a quoting round trip, a custom JSON encoder, JSON Lines validated into dataclasses, packed records verified by a digest, byte orders and base64, an inventory in SQLite and a sales join. The checkpoint adds a CSV-to-JSON converter, a binary record store with a SHA-256 check, and a grouped SQLite report with a rolled-back transaction.

CONTENTS

1 Reading and writing files — open, modes, encoding and with 13 min

2	pathlib — paths as objects	13 min
3	CSV — reader, writer, DictReader and the quoting rules	13 min
4	JSON in depth — custom encoders, decoders, dataclasses and config files	13 min
5	Bytes and binary data — struct, int.to_bytes, base64 and hashlib	13 min
6	sqlite3 — a SQL database in the standard library	14 min
✓	Checkpoint — Files and data formats	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Reading and writing files — open, modes, encoding and with

`open` returns a file object, and everything about files in Python follows from three choices made in that call: the *mode* (read, write, append; text or binary), the *encoding* (which should always be stated for text), and whether the object is used inside a `with` block (it should be). This lesson covers those choices, the read and write methods and when each is right, line iteration and the newline rules, `print(file=)`, in-memory files with `io.StringIO`, and the facts that matter for the judge — a program may create and read files in its working directory — and for real systems, where a file that is not closed is a resource leak and a partly written file is data loss.

---

open

```
with open("notes.txt", "w", encoding="utf-8") as f:
    f.write("first line\n")
    f.write("second line\n")

with open("notes.txt", encoding="utf-8") as f:      # mode "r" is the default
    text = f.read()
```

Mode	Meaning
`"r"`	read text (default); `FileNotFoundError` if absent
`"w"`	write text; **truncates** an existing file or creates it
`"a"`	append text; creates if absent
`"x"`	create for writing; `FileExistsError` if present
`"r+"`	read and write, no truncation
`"rb"`, `"wb"`, `"ab"`	the same in binary: `bytes` in, `bytes` out

Text mode decodes and encodes with `encoding` and translates line endings (`\r\n` on Windows becomes `\n` on read, and `\n` becomes the platform's on write — `newline=""`

disables that, which `csv` requires). Binary mode does neither. State `encoding="utf-8"` on every text open: the default is the platform's preferred encoding, which is not UTF-8 on every Windows machine, and a file written on one system is then unreadable on another.

---

with

`open` acquires a file descriptor — a limited OS resource — and buffers writes in memory. `with` guarantees `close()` on every exit (Module 10), which flushes the buffer to disk and releases the descriptor. Without it, a write may sit in the buffer until the interpreter exits, and a loop that opens files without closing them runs out of descriptors. The rule has no exceptions worth learning.

Reading

```
with open(path, encoding="utf-8") as f:
    everything = f.read()                  # the whole file as one string
    # — or —
    lines = f.readlines()                  # a list of lines, each with its "\r
    # — or —
    for line in f:                         # one line at a time, lazily — the i
        line = line.rstrip("\n")
    # — or —
    first = f.readline()                   # one line; "" at end of file
    chunk = f.read(4096)                   # up to n characters
```

Iterating the file object is the right default: it reads as it goes and never holds more than one line, so a multi-gigabyte log costs the memory of its longest line. `read()` is right for small files that are processed as a whole (a config, a template); `read(n)` in a loop for binary streams. After a full read the position is at the end; `f.seek(0)` rewinds.

Writing

```
with open(path, "w", encoding="utf-8") as f:
    f.write("one line\n")                          # write adds no newline
    f.writelines(f"{x}\n" for x in xs)             # an iterable of strings, n
    print("via print", file=f)                     # print adds the newline an
    print(*values, sep=",", file=f)
```

`write` takes a string and returns the count written; it does not add a newline. `print(..., file=f)` is often the most convenient for line-oriented output because it formats and terminates. Writing is buffered; `f.flush()` pushes the buffer without closing (for progress files another process reads).

The safe way to replace a file is to write a new one and rename it over the old, so a crash mid-write leaves the original intact: `tempfile.NamedTemporaryFile(dir=..., delete=False)`, write, close, `os.replace(tmp, path)`. Overwriting in place with `"w"` truncates first and loses the data if the write fails.

Paths, briefly

The path passed to `open` is relative to the *current working directory* — wherever the process was started, which is not necessarily the script's directory. Build paths from a known base (`pathlib.Path(__file__).parent`, next lesson) rather than assuming. Forward slashes work on every platform. On the judge the working directory is writable, and a program may create, write, read and delete files there — which is how the file exercises in this module work; they leave nothing behind.

In-memory files

```
import io

buf = io.StringIO()
print("captured", file=buf)
buf.getvalue()                    # 'captured\n'

reader = io.StringIO("a,b\n1,2\n")
for line in reader:               # behaves like an open text file
    ...
bio = io.BytesIO(b"\x00\x01")     # the bytes equivalent
```

`StringIO` and `BytesIO` are file objects backed by memory: the way to test code that takes a file, to capture output, and to feed `csv` / `json` from a string. Anything that accepts "a file-like object" accepts them.

Errors

`FileNotFoundError`, `PermissionError`, `IsADirectoryError` are subclasses of `OSError` and carry `.filename`; `UnicodeDecodeError` means the encoding was wrong (`errors="replace"` substitutes U+FFFD, `errors="ignore"` drops, both hide the problem). EAFP applies: `try: open(...)` rather than `if os.path.exists(...)`, because the file can vanish between the check and the open.

Pitfalls

No `encoding=` on a text open.

A file opened without `with` — unflushed writes, leaked descriptors, `ResourceWarning`.

`"w"` when `"a"` was meant: the file is truncated.

`read()` on a huge file when iteration would do.

`readlines()` then indexing, when a loop was meant; and forgetting the `\n` on each element.

A relative path that depends on the working directory.

Key takeaways

`open(path, mode, encoding="utf-8")` inside `with`; modes `r` / `w` / `a` / `x` and `b` for binary; `w` truncates.

Iterate the file object for lines; `read()` for small whole files; `readline()` returns `""` at the end.

`write` adds nothing; `print(file=f)` formats and terminates; writes are buffered until `flush`/close.

Replace files by writing a temporary and `os.replace`-ing it; paths are relative to the working directory.

`io.StringIO` / `BytesIO` are in-memory files for tests and captured output; file errors are `OSError` subclasses — use EAFP.

EXERCISE 1

Write, then read back

Read lines from standard input and write them to a file `out_demo.txt` (UTF-8, one line each, inside a `with`). Print the file's size in bytes (`os.path.getsize`). Then reopen it for reading, iterate its lines and print each numbered from 1 with the newline stripped, followed by `lines <n>`. Delete the file at the end (in a `finally`).

Input: zero or more lines. **Output:** `bytes <n>`, then `<i>: <line>` per line, then `lines <n>`.

```
héllo
world
```

prints

```
bytes 13
1: héllo
2: world
lines 2
```

INPUT 1

```
héllo
world
```

OUTPUT 1

```
bytes 13
1: héllo
2: world
lines 2
```

REFERENCE SOLUTION

```
import os
import sys

PATH = "out_demo.txt"
lines = [line.rstrip("\n") for line in sys.stdin]
try:
    with open(PATH, "w", encoding="utf-8") as f:
        for line in lines:
            f.write(line + "\n")
    print(f"bytes {os.path.getsize(PATH)}")
    count = 0
    with open(PATH, encoding="utf-8") as f:
        for i, line in enumerate(f, start=1):
            print(f"{i}: {line.rstrip(chr(10))}")
            count += 1
    print(f"lines {count}")
finally:
    if os.path.exists(PATH):
        os.remove(PATH)
```

EXERCISE 2

An append-only log

Maintain `app_demo.log` from commands: `log <text>` appends one line using `print(..., file=f)` on a file opened in append mode; `show` prints the file's lines numbered; `count` prints the number of lines by iterating the file; `clear` truncates it by opening in `"w"` mode. Delete the file at the end.

Input: commands. **Output:** the lines of `show` and `count`.

```
log first
log second, with comma
show
count
clear
count
```

prints

```
1: first
2: second, with comma
2
0
```

INPUT 1

```
log first
log second, with comma
show
count
clear
count
```

OUTPUT 1

```
1: first
2: second, with comma
2
0
```

REFERENCE SOLUTION

```python
import os
import sys

PATH = "app_demo.log"
open(PATH, "w", encoding="utf-8").close()   # start empty
try:
    for line in sys.stdin:
        cmd, _, text = line.rstrip("\n").partition(" ")
        if cmd == "log":
            with open(PATH, "a", encoding="utf-8") as f:
                print(text, file=f)
        elif cmd == "show":
            with open(PATH, encoding="utf-8") as f:
                for i, entry in enumerate(f, start=1):
                    print(f"{i}: {entry.rstrip(chr(10))}")
        elif cmd == "count":
            with open(PATH, encoding="utf-8") as f:
                print(sum(1 for _ in f))
        elif cmd == "clear":
            open(PATH, "w", encoding="utf-8").close()
finally:
    os.remove(PATH)
```

QUIZ

1. What does opening an existing file with mode `"w"` do?

A. Appends

B. Truncates it to empty before writing

C. Raises `FileExistsError`

D. Opens read-only

2. Why pass `encoding="utf-8"` to `open` for text?

A. It is required syntax

B. The default is the platform's preferred encoding, which differs between systems

C. It makes reading faster

D. Only for binary files

3. Which reads a large file with the least memory?

A. `f.read()`

B. `f.readlines()`

C. `for line in f:`

D. `list(f)`

4. What guarantees a file is closed and its buffer flushed?

A. Garbage collection

B. The `with` statement

C. Calling `read()`

D. Nothing is needed

5. What is `io.StringIO` for?

A. Reading binary files

B. An in-memory file object: capturing output or testing code that takes a file

C. Formatting strings

D. Encoding text

ANSWER KEY · READING AND WRITING FILES — OPEN, MODES, ENCODING AND WITH

1. B `"a"` appends; `"x"` refuses to overwrite; `"w"` discards the old contents.

2. B A file written with one default and read with another produces `UnicodeDecodeError` or mojibake.

3. C Iterating the file object yields one line at a time; the others load everything.

4. B `__exit__` calls `close()` on every path; relying on GC is unreliable and warns under `-X dev`.

5. B Anything that accepts a file-like object accepts it; `getvalue()` returns what was written.

LESSON 2 · 13 MIN

pathlib — paths as objects

A path is not a string. It has a directory part, a name, a stem and a suffix; it can be joined, made absolute, tested for existence, listed, created and removed; and the separator differs between operating systems. `pathlib.Path` (3.4) models all of that as an object with methods, replacing the string juggling of `os.path` with `/` for joining and readable names for everything else. This lesson covers constructing and joining paths, the parts, the queries, reading and writing through a path, listing and searching a tree, creating and deleting, and the pure-path classes that manipulate paths without touching the file system.

Constructing and joining

```
from pathlib import Path

p = Path("data") / "reports" / "q1.csv"       # / joins, on every platform
p                                             # PosixPath('data/reports/q1.csv
str(p)                                        # 'data/reports/q1.csv' — for AP
Path.cwd(), Path.home()                       # the working directory, the hom
Path(__file__).parent                         # the directory this script live
p.resolve()                                   # absolute, symlinks resolved
Path("a/b/../c")                              # not simplified; resolve() does
```

`Path("dir") / "file"` is the idiom; the operator takes strings or paths on the right. Paths are immutable and hashable, so they work as dict keys and set members. Nearly every function that takes a filename (`open` , `os.remove` , `shutil.copy` , `json.load` via `open`) accepts a `Path` .

The parts

```
p = Path("/home/ada/data/report.tar.gz")
p.name          # 'report.tar.gz'
p.stem          # 'report.tar'
p.suffix        # '.gz'
p.suffixes      # ['.tar', '.gz']
p.parent        # PosixPath('/home/ada/data')
p.parents[1]    # PosixPath('/home/ada')
p.parts         # ('/', 'home', 'ada', 'data', 'report.tar.gz')
p.anchor        # '/'
p.with_suffix(".zip")        # PosixPath('/home/ada/data/report.tar.zip') — re
p.with_name("summary.txt")
p.with_stem("report2")       # 3.9
p.is_absolute()
p.relative_to("/home/ada")   # PosixPath('data/report.tar.gz'); ValueError if
```

`stem` and `suffix` split at the last dot, which is right for `report.csv` and surprising for `report.tar.gz`; `suffixes` has them all.

Queries

```
p.exists(), p.is_file(), p.is_dir(), p.is_symlink()
p.stat().st_size                # bytes
p.stat().st_mtime               # modification time (a timestamp)
p.samefile(other)
```

These touch the file system; the ones above did not. As always, a check followed by an action has a window; prefer trying the action and catching `OSError` when the outcome matters.

Reading and writing through a path

```
text = p.read_text(encoding="utf-8")          # the whole file
p.write_text("hello\n", encoding="utf-8")     # creates or truncates
data = p.read_bytes(); p.write_bytes(b"...")
with p.open(encoding="utf-8") as f:           # the same as open(p, ...)
    for line in f:
        ...
```

`read_text` / `write_text` are the two-liners for small files; `open` remains the tool for streaming. State the encoding here too.

Listing and searching

```
for child in sorted(Path("data").iterdir()):     # direct children, files and
    print(child.name)
list(Path("data").glob("*.csv"))                  # matching names in one dire
list(Path("data").rglob("*.csv"))                 # recursively, every depth
list(Path("data").glob("**/*.csv"))               # the same as rglob
[p for p in Path("src").rglob("*.py") if "test" not in p.parts]
```

`iterdir`, `glob` and `rglob` yield in *file-system order*, which is arbitrary — sort before printing or relying on it. Patterns are shell-style (`*`, `?`, `[abc]`), and `rglob` walks directories, which can be slow on large trees; `os.walk` is the lower-level alternative that yields `(dirpath, dirnames, filenames)` and lets you prune `dirnames` in place.

Creating, moving, deleting

```
Path("out/logs").mkdir(parents=True, exist_ok=True)   # like mkdir -p
p.touch()                                              # create empty or updat
p.rename("new_name.txt")                               # or replace() to overw
p.unlink(missing_ok=True)                              # delete a file (3.8 fo
Path("empty_dir").rmdir()                              # only if empty
import shutil
shutil.rmtree("tree")                                  # delete recursively —
shutil.copy(src, dst); shutil.move(src, dst)
```

`mkdir(parents=True, exist_ok=True)` is the form that never raises for an existing directory; `rmtree` is the one call to think twice about.

Pure paths

`PurePosixPath` and `PureWindowsPath` do the parsing, joining and parts without ever touching a disk — for manipulating paths *for* another system, or in tests that must not depend on the file system:

```
from pathlib import PurePosixPath, PureWindowsPath
PurePosixPath("/srv/app") / "logs" / "x.log"          # PurePosixPath('/srv/ap
PureWindowsPath(r"C:\Users\ada").parts                 # ('C:\\', 'Users', 'ad
```

os.path, for reading old code

`os.path.join(a, b)`, `os.path.basename`, `dirname`, `splitext`, `exists`, `abspath` are the string-based ancestors; each has a `Path` equivalent (`/`, `.name`, `.parent`, `.stem` / `.suffix`, `.exists()`, `.resolve()`). New code uses `pathlib`; `os.path` still appears everywhere and reads easily once the mapping is known.

A worked example: renaming by pattern

```
for path in sorted(Path("photos").glob("IMG_*.jpg")):
    number = path.stem.removeprefix("IMG_")
    target = path.with_name(f"holiday_{int(number):04d}{path.suffix}")
    if not target.exists():
        path.rename(target)
```

Every step is a path operation: `glob` finds the candidates, `stem` and `suffix` take the name apart, `with_name` builds the new one beside the old, `exists` guards against clobbering, `rename` moves. No string slicing, no separators, and the same code runs on every platform.

Pitfalls

- Building paths with `+` and hard-coded slashes.
- `p.suffix` on a double extension.
- Relying on the order of `iterdir` / `glob` — sort.
- `mkdir` without `parents=True, exist_ok=True` in a setup step that may run twice.
- `rmtree` on a path built from user input without checking what it resolves to.
- Passing a `Path` to a library that insists on `str` — `str(p)` or `os.fspath(p)`.

Key takeaways

`Path(...) / "part"` joins; `.name`, `.stem`, `.suffix`, `.parent`, `.parts`, `.with_suffix` decompose and rebuild.

`.exists()`, `.is_file()`, `.stat()` query; `.read_text` / `.write_text` (with an encoding) and `.open()` do I/O.

`iterdir`, `glob`, `rglob` list — sort the results; `mkdir(parents=True, exist_ok=True)`, `unlink`, `rename`, `shutil.rmtree` change the tree.

Pure paths manipulate without touching the disk.

Paths are immutable, hashable and accepted wherever a filename is.

EXERCISE 1

Path parts

Read POSIX path strings, one per line, and decompose each with `PurePosixPath` (no file system involved): print `name`, `stem`, `suffix`, all `suffixes` joined with `+` (or `-` if none), `parent`, and the path with its suffix replaced by `.bak`.

Input: lines. **Output:** `name=<n> stem=<s> suffix=<x> suffixes=<a+b> parent=<p> bak=<q>` per line.

```
/home/ada/data/report.tar.gz
```

prints

```
name=report.tar.gz stem=report.tar suffix=.gz suffixes=.tar+.gz parent=/home/ada/dat
```

INPUT 1

```
/home/ada/data/report.tar.gz
```

OUTPUT 1

```
name=report.tar.gz stem=report.tar
suffix=.gz suffixes=.tar+.gz
parent=/home/ada/data
bak=/home/ada/data/report.tar.bak
```

REFERENCE SOLUTION

```python
import sys
from pathlib import PurePosixPath

for line in sys.stdin:
    p = PurePosixPath(line.strip())
    suffixes = "+".join(p.suffixes) if p.suffixes else "-"
    print(f"name={p.name} stem={p.stem} suffix={p.suffix} suffixes={suffixes} parent
```

EXERCISE 2

Build a tree and search it

Create a directory `tree_demo` next to this script and populate it from `mk <relative/path>` lines (creating parent directories with `mkdir(parents=True, exist_ok=True)` and empty files with `touch`). Then for each `find <pattern>` line print the matching paths from `rglob` relative to the root, sorted, one per line (or `none`), and for `count` print the number of files in the tree. Remove the tree at the end with `shutil.rmtree`.

Input: commands. **Output:** the results of `find` and `count`.

```
mk docs/a.md
mk docs/sub/b.md
mk src/main.py
find *.md
count
```

prints

```
docs/a.md
docs/sub/b.md
3
```

INPUT 1

```
mk docs/a.md
mk docs/sub/b.md
mk src/main.py
find *.md
count
```

OUTPUT 1

```
docs/a.md
docs/sub/b.md
3
```

REFERENCE SOLUTION

```
import shutil
import sys
from pathlib import Path

root = Path(__file__).resolve().parent / "tree_demo"
root.mkdir(exist_ok=True)
try:
    for line in sys.stdin:
        cmd, *args = line.split()
        if cmd == "mk":
            target = root / args[0]
            target.parent.mkdir(parents=True, exist_ok=True)
            target.touch()
        elif cmd == "find":
            hits = sorted(p.relative_to(root).as_posix() for p in root.rglob(args[0]
            print("\n".join(hits) if hits else "none")
        elif cmd == "count":
            print(sum(1 for p in root.rglob("*") if p.is_file()))
finally:
    shutil.rmtree(root, ignore_errors=True)
```

QUIZ

1. What does `Path("a") / "b" / "c.txt"` produce?

A. A string

B. `Path('a/b/c.txt')` with the platform's separator

C. An error

D. `Path('a')`

2. What is `Path("x.tar.gz").suffix`?

A. `'.tar.gz'`

B. `'.gz'`

C. `'.tar'`

D. `''`

3. Which creates `out/logs` including missing parents without failing if it exists?

A. `Path("out/logs").mkdir()`

B. `Path("out/logs").mkdir(parents=True, exist_ok=True)`

C. `Path("out/logs").touch()`

D. `os.mkdir("out/logs")`

4. What order does `Path.iterdir()` yield in?

A. Alphabetical

B. File-system order — arbitrary; sort it

C. Creation order

D. Size order

5. What are `PurePosixPath` and `PureWindowsPath` for?

A. Faster I/O

B. Manipulating paths without touching the file system, including paths for another OS

C. Network paths

D. They are deprecated

ANSWER KEY · PATHLIB — PATHS AS OBJECTS

1. B The `/` operator joins path components portably.

2. B `suffix` is the last extension; `suffixes` gives `['.tar', '.gz']`.

3. B Without the flags, a missing parent or an existing directory raises.

4. B Judged and tested output must not depend on directory order.

5. B Pure paths parse, join and decompose; the concrete `Path` adds the I/O methods.

LESSON 3 · 13 MIN

CSV — reader, writer, DictReader and the quoting rules

Comma-separated values look simple enough to split by hand, and every hand-written CSV parser breaks on the first field that contains a comma, a quote or a newline. The `csv` module handles the quoting rules, the dialects (tab-separated, semicolons, Excel's conventions) and the line-ending trap, and its `DictReader` / `DictWriter` map rows to dictionaries keyed by the header. This lesson covers reading rows as lists and as dicts, writing with the correct `newline=""`, delimiters and quoting, computing over columns with the right conversions, reading from stdin and strings, and the cases CSV cannot represent.

Reading

```
import csv

with open("sales.csv", newline="", encoding="utf-8") as f:
    reader = csv.reader(f)
    header = next(reader)                       # the first row, as a list of
    for row in reader:                          # each row: a list of strings
        name, qty, price = row
```

Every field is a *string* — `"3"` and `"2.50"` must be converted. `newline=""` on open is required by the module: it leaves line endings to `csv`, which knows that a quoted field may contain one. The reader handles `"a, b"` (a quoted field with a comma), `"say ""hi"""` (a doubled quote inside a quoted field), and empty fields.

DictReader and DictWriter

```
with open("sales.csv", newline="", encoding="utf-8") as f:
    for row in csv.DictReader(f):                # header row → keys
        total += int(row["qty"]) * float(row["price"])

with open("out.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["name", "total"])
    writer.writeheader()
    writer.writerow({"name": "ada", "total": 12.5})
    writer.writerows(rows)                       # an iterable of dicts
```

`DictReader` takes the first row as field names (or `fieldnames=` when the file has no header) and yields one dict per row in file order; missing trailing fields are `None`, extra ones go under `restkey`. `DictWriter` needs `fieldnames` to fix the column order and raises on a dict with an unknown key (`extrasaction="ignore"` to drop). Rows by name survive column reordering; rows by position do not.

Writing

```
with open("out.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["name", "note"])
    writer.writerow(["ada", "says, \"hi\""])   # quoted and escaped automatic
```

The writer quotes only fields that need it (`QUOTE_MINIMAL`); `quoting=csv.QUOTE_ALL` quotes everything, `QUOTE_NONNUMERIC` quotes strings and leaves numbers bare (and makes the *reader* convert unquoted fields to floats). Values are converted with `str()`, so `None` becomes an empty string.

Delimiters and dialects

```
csv.reader(f, delimiter="\t")                    # TSV
csv.reader(f, delimiter=";")                     # European Excel
csv.reader(f, dialect="excel-tab")
csv.Sniffer().sniff(sample).delimiter            # guess from a sample
```

A dialect bundles delimiter, quote character, escaping and line terminator; `excel` is the default and matches what spreadsheets produce. When a file has a header you know, prefer

stating the delimiter to sniffing it.

From stdin and from strings

```
import sys
for row in csv.reader(sys.stdin):                # stdin is a file object; no r
    ...

import io
rows = list(csv.reader(io.StringIO("a,b\n1,2\n")))      # [['a', 'b'], ['1',
text = "x,y\n"
csv.reader(text.splitlines())                             # any iterable of li
```

The reader accepts any iterable of strings, which is what makes it testable with a list and usable on the judge with `sys.stdin`.

Computing over columns

```
from collections import defaultdict
totals = defaultdict(float)
for row in csv.DictReader(f):
    totals[row["region"]] += int(row["qty"]) * float(row["price"])
for region in sorted(totals):
    print(f"{region},{totals[region]:.2f}")
```

Convert at the edge (`int`, `float`, `Decimal` for money, `date.fromisoformat` for dates), validate with `try` / `except ValueError` per row (Module 5), aggregate with the collections of Module 7, and format the output with fixed decimals. A CSV of a few hundred thousand rows is fine to stream this way; larger analyses move to `pandas`, which reads the same files with `read_csv`.

Headers that vary

Real files arrive with columns in different orders, with extra columns, or with a header that differs in case or spacing from what the code expects. `DictReader` absorbs order changes; for the rest, normalise the header once — `reader.fieldnames = [h.strip().lower() for h`

`in reader.fieldnames]` — and check that the required names are present before the loop, raising a clear error naming the missing column. A row count and a bad-row count printed at the end turn a silent partial import into a visible one.

What CSV cannot do

There is no type information — everything is text until you convert it; there is no standard for dates, booleans or nulls (an empty field could be any of them); nested data does not fit; and encodings are unspecified (a file from Excel may be Windows-1252 or UTF-8 with a byte-order mark — `encoding="utf-8-sig"` strips the mark). For structured, typed or nested data, JSON (next lesson) is the better exchange format; CSV's strength is that every spreadsheet and database can produce and consume it.

Pitfalls

- Splitting on commas by hand.
- Forgetting `newline=""` on the file used for writing (blank lines between rows on Windows).
- Treating fields as numbers without converting.
- A `DictWriter` without `fieldnames`, or with a row containing an unexpected key.
- Reading an Excel export as `utf-8` and getting `` in the first header.
- Assuming an empty field means zero.

Key takeaways

- `csv.reader` / `writer` for rows as lists, `DictReader` / `DictWriter` for rows as dicts keyed by the header.
- Open with `newline=""` and an encoding; every field read is a string — convert at the edge.
- The writer quotes and escapes for you; `delimiter=` and dialects handle TSV and semicolons.
- The reader takes any iterable of lines: files, `sys.stdin`, `StringIO`, lists.
- CSV carries no types or nesting; JSON for structured data, `pandas` for large analyses.

EXERCISE 1

Totals by category

Read a CSV document from standard input with `csv.DictReader` — columns `name`, `category`, `qty`, `price` — and total `qty * price` per category. Write the result as CSV to standard output with `csv.writer(sys.stdout)`: a header `category,total` then one row per category in name order, totals with two decimals.

Input: a CSV document. **Output:** a CSV document.

```
name,category,qty,price
bolt,hardware,10,0.5
nut,hardware,4,0.25
glue,supplies,1,3
```

prints

```
category,total
hardware,6.00
supplies,3.00
```

INPUT 1

```
name,category,qty,price
bolt,hardware,10,0.5
nut,hardware,4,0.25
glue,supplies,1,3
```

OUTPUT 1

```
category,total
hardware,6.00
supplies,3.00
```

REFERENCE SOLUTION

```python
import csv
import sys
from collections import defaultdict

totals = defaultdict(float)
for row in csv.DictReader(sys.stdin):
    totals[row["category"]] += int(row["qty"]) * float(row["price"])
writer = csv.writer(sys.stdout, lineterminator="\n")
writer.writerow(["category", "total"])
for category in sorted(totals):
    writer.writerow([category, f"{totals[category]:.2f}"])
```

EXERCISE 2

Quoting round trip

Each input line is `name|note` where the note may contain commas and double quotes. Write the pairs to `notes_demo.csv` with `csv.writer` (`newline=""`), then print the file's **raw** lines as `repr` strings to show how the writer quoted them, then read the file back with `csv.reader` and print `name → note` per row. Delete the file at the end.

Input: lines. **Output:** the raw lines as reprs, then the parsed rows.

```
ada|says, "hi"
bob|plain
```

prints

```
'ada,"says, ""hi"""'
'bob,plain'
ada → says, "hi"
bob → plain
```

INPUT 1

```
ada|says, "hi"
bob|plain
```

OUTPUT 1

```
'ada,"says, ""hi"""'
'bob,plain'
ada → says, "hi"
bob → plain
```

REFERENCE SOLUTION

```python
import csv
import os
import sys

PATH = "notes_demo.csv"
pairs = [line.rstrip("\n").split("|", 1) for line in sys.stdin if line.strip()]
try:
    with open(PATH, "w", newline="", encoding="utf-8") as f:
        csv.writer(f, lineterminator="\n").writerows(pairs)
    with open(PATH, encoding="utf-8") as f:
        for raw in f:
            print(repr(raw.rstrip("\n")))
    with open(PATH, newline="", encoding="utf-8") as f:
        for name, note in csv.reader(f):
            print(f"{name} → {note}")
finally:
    if os.path.exists(PATH):
        os.remove(PATH)
```

QUIZ

1. Why open a CSV file with `newline=""`?

A. To speed up reading

B. So the `csv` module controls line endings — quoted fields may contain newlines, and writers avoid doubled `\r\n`

C. It is required by `open`

D. To disable quoting

2. What type is a field returned by `csv.reader`?

A. Whatever it looks like

B. Always `str`

C. `int` for digits

D. `bytes`

3. What does `DictWriter` require?

A. Nothing

B. `fieldnames`, to fix the column order

C. A header already in the file

D. Sorted rows

4. How does a CSV writer represent the field `say "hi"`?

A. `say \"hi\"`

B. `"say ""hi"""` — quoted, inner quotes doubled

C. Unquoted

D. It raises

5. Which encoding strips an Excel byte-order mark on read?

A. `utf-8`

B. `utf-8-sig`

C. `ascii`

D. `latin-1`

ANSWER KEY · CSV — READER, WRITER, DICTREADER AND THE QUOTING RULES

1. B The module's documentation requires it for both reading and writing.

2. B CSV carries no types; convert at the edge.

3. B `writeheader()` then writes them; a row with an unknown key raises unless `extrasaction="ignore"`.

4. B Doubling is the CSV escape; the reader reverses it.

5. B Without it, the first header name starts with `\ufeff`.

LESSON 4 · 13 MIN

JSON in depth — custom encoders, decoders, dataclasses and config files

Module 7 covered `json.loads` / `dumps` and the mapping between JSON and Python values. This lesson is about the edges: the objects JSON cannot serialise and the `default=` hook and `JSONEncoder` subclass that teach it to; `object_hook` and `object_pairs_hook` for turning parsed dicts back into your types; dataclasses in and out via `asdict`; dates as ISO strings; `JSONDecodeError` and validation; JSON Lines for streams; and the two other configuration formats in the standard library, TOML via `tomllib` and INI via `configparser`, with the rule for choosing among them.

Serialising what JSON does not know

```
import json
from datetime import date
from decimal import Decimal

json.dumps({"when": date(2024, 5, 1)})          # TypeError: Object of type da

def to_json(o):
    if isinstance(o, date):
        return o.isoformat()
    if isinstance(o, Decimal):
        return str(o)
    if isinstance(o, set):
        return sorted(o)
    raise TypeError(f"not serializable: {type(o).__name__}")

json.dumps({"when": date(2024, 5, 1), "tags": {"b", "a"}}, default=to_json)
# '{"when": "2024-05-01", "tags": ["a", "b"]}'
```

`default` is called for any object the encoder cannot handle and must return something it can; raising `TypeError` for the rest keeps mistakes loud. A `JSONEncoder` subclass with `default(self, o)` is the class-based form, passed as `cls=`. Sets become sorted lists (deterministic and valid JSON), dates become ISO strings, `Decimal` becomes a string to preserve its exactness (a number would round-trip as a float).

Dataclasses

```
from dataclasses import dataclass, asdict

@dataclass
class Item:
    name: str
    qty: int

json.dumps(asdict(Item("bolt", 3)))            # '{"name": "bolt", "qty": 3}'
json.dumps([asdict(i) for i in items], indent=2)
Item(**json.loads('{"name": "nut", "qty": 5}'))  # back again
```

`asdict` produces a plain dict (recursively), and `Cls(**d)` rebuilds an instance — the whole round trip for flat records. For nested dataclasses, rebuild inside out, or use a `from_dict` class method that knows the shape. Third-party libraries (`pydantic`, `attrs`, `marshmallow`) automate this with validation; the standard library gives you the two calls.

Decoding into your types

```
def as_item(d):
    if d.keys() == {"name", "qty"}:
        return Item(d["name"], d["qty"])
    return d

json.loads(text, object_hook=as_item)          # every JSON object is passed t
json.loads(text, object_pairs_hook=OrderedDict)   # receives the pairs in orde
json.loads("1.5", parse_float=Decimal)         # numbers as Decimal instead of
```

`object_hook` sees every object in the document, bottom-up, so it must return the dict unchanged when it does not recognise the shape. `parse_float=Decimal` is the way to read money from JSON without float error.

Validation

`json.loads` guarantees syntax, not shape. After parsing, check the shape once at the boundary — with `match` patterns (Module 3), explicit tests, or a dataclass `__post_init__` —

and raise a clear error:

```
try:
    doc = json.loads(text)
except json.JSONDecodeError as e:
    raise ValueError(f"invalid JSON at line {e.lineno}, column {e.colno}: {e.r
match doc:
    case {"name": str(name), "qty": int(qty)} if qty ⩾ 0:
        ...
    case _:
        raise ValueError("expected {name: str, qty: int ⩾ 0}")
```

`JSONDecodeError` carries `lineno`, `colno` and `msg`; it is a subclass of `ValueError`, so a broad `except ValueError` catches it too.

Deterministic and pretty

`sort_keys=True` makes equal dicts serialise identically (Module 7); `indent=2` is for humans; `separators=(",", ":")` is the compact form for storage and transmission; `ensure_ascii=False` keeps non-ASCII characters readable. `python -m json.tool file.json` pretty-prints from the command line.

JSON Lines

One JSON value per line — `{"id": 1}\n{"id": 2}\n` — is the format for logs and streams: appendable, splittable, readable one record at a time with `for line in f: record = json.loads(line)`, and writable with `f.write(json.dumps(record) + "\n")`. A single huge JSON array cannot be processed until it has been read entirely; JSON Lines can.

TOML and INI

```
import tomllib
with open("pyproject.toml", "rb") as f:          # tomllib wants bytes
    cfg = tomllib.load(f)
cfg["project"]["name"]

import configparser
parser = configparser.ConfigParser()
parser.read("settings.ini")
parser["server"]["port"]                         # always a string
parser["server"].getint("port")
```

TOML (3.11, read-only in the standard library) is typed — strings, ints, floats, booleans, dates, arrays, tables — and is the format for `pyproject.toml` and modern configuration. INI via `configparser` is older, untyped (everything is a string, with `getint` / `getboolean` helpers) and still common. The rule: JSON for data exchanged with programs, TOML for configuration written by people, INI when a system already uses it, and never `pickle` for anything that crosses a trust boundary.

Large documents

`json.load` parses the whole document into memory — several times its byte size, since every value becomes an object. For a file of hundreds of megabytes that is the wrong tool: switch the producer to JSON Lines and stream it line by line, or use an incremental parser (`ijson`, third-party) that yields values as it reads. The same reasoning applies to `json.dumps` of a huge structure: write records one at a time rather than building one giant string.

Pitfalls

- `json.dumps` on a set, date, `Decimal` or dataclass without `default=` / `asdict`.
- Reading money as floats — `parse_float=Decimal`.
- Trusting shape after `loads`; validate at the boundary.
- `tomllib.load` on a text-mode file (it wants `"rb"`).
- Expecting `configparser` values to be numbers.
- A single giant JSON array where JSON Lines would stream.

Key takeaways

- `default=` (or a `JSONEncoder` subclass) serialises dates as ISO strings, `Decimal` as strings, sets as sorted lists.
- `asdict` and `Cls(**d)` round-trip flat dataclasses; `object_hook` and `parse_float` shape the decode.
- `JSONDecodeError` has line and column; validate the shape after parsing with `match` or explicit checks.
- `sort_keys` for determinism, `indent` for people, compact separators for storage; JSON Lines for streams.
- `tomllib` (bytes, read-only) for typed configuration; `configparser` for INI; JSON for exchange.

EXERCISE 1

A custom encoder

Build a document from typed lines — `date <iso>`, `set <words…>`, `decimal <number>`, `int <n>`, `text <words…>` — storing each under its key (the first word; a repeated key overwrites) as a real `date`, `set`, `Decimal`, `int` or `str`. Serialise it with `json.dumps(doc, default=to_json, sort_keys=True)` where `to_json` renders dates as ISO strings, sets as sorted lists and decimals as strings, and raises `TypeError` for anything else.

Input: lines. **Output:** one JSON line.

```
date 2024-05-01
set b a c
decimal 1.50
int 7
text hello world
```

prints

```
{"date": "2024-05-01", "decimal": "1.50", "int": 7, "set": ["a", "b", "c"], "text":
```

INPUT 1

```
date 2024-05-01
set b a c
decimal 1.50
int 7
text hello world
```

OUTPUT 1

```
{"date": "2024-05-01", "decimal":
"1.50", "int": 7, "set": ["a", "b",
"c"], "text": "hello world"}
```

REFERENCE SOLUTION

```python
import json
import sys
from datetime import date
from decimal import Decimal


def to_json(o):
    if isinstance(o, date):
        return o.isoformat()
    if isinstance(o, set):
        return sorted(o)
    if isinstance(o, Decimal):
        return str(o)
    raise TypeError(f"not serializable: {type(o).__name__}")


doc = {}
for line in sys.stdin:
    kind, _, rest = line.rstrip("\n").partition(" ")
    if kind == "date":
        doc[kind] = date.fromisoformat(rest)
    elif kind == "set":
        doc[kind] = set(rest.split())
    elif kind == "decimal":
        doc[kind] = Decimal(rest)
    elif kind == "int":
        doc[kind] = int(rest)
    elif kind == "text":
        doc[kind] = rest
print(json.dumps(doc, default=to_json, sort_keys=True))
```

EXERCISE 2

JSON Lines into dataclasses

Each input line is a JSON document meant to be `{"name": str, "qty": int}` with `qty ⩾ 0`. Parse each with `json.loads` (a `JSONDecodeError` counts as invalid), validate the shape with a

`match` pattern, build a frozen `Item` dataclass for the valid ones, and at the end print the valid items as a JSON array of `asdict` results with `sort_keys=True`, then `invalid <n>`.

Input: JSON lines. **Output:** one JSON line, then `invalid <n>`.

```
{"name": "bolt", "qty": 3}
{"name": "nut", "qty": -1}
not json
{"qty": 2, "name": "glue", "extra": true}
```

prints

```
[{"name": "bolt", "qty": 3}, {"name": "glue", "qty": 2}]
invalid 2
```

INPUT 1

```
{"name": "bolt", "qty": 3}
{"name": "nut", "qty": -1}
not json
{"qty": 2, "name": "glue", "extra":
true}
```

OUTPUT 1

```
[{"name": "bolt", "qty": 3}, {"name":
"glue", "qty": 2}]
invalid 2
```

REFERENCE SOLUTION

```python
import json
import sys
from dataclasses import asdict, dataclass


@dataclass(frozen=True)
class Item:
    name: str
    qty: int


items, invalid = [], 0
for line in sys.stdin:
    if not line.strip():
        continue
    try:
        doc = json.loads(line)
    except json.JSONDecodeError:
        invalid += 1
        continue
    match doc:
        case {"name": str(name), "qty": int(qty)} if qty ⩾ 0 and not isinstance(qty
            items.append(Item(name, qty))
        case _:
            invalid += 1
print(json.dumps([asdict(i) for i in items], sort_keys=True))
print(f"invalid {invalid}")
```

QUIZ

1. What does `json.dumps(date.today())` do without help?

A. Produces an ISO string

B. Raises `TypeError: Object of type date is not JSON serializable`

C. Produces a timestamp

D. Produces `null`

2. How do you read JSON numbers as `Decimal`?

A. `json.loads(s, decimal=True)`

B. `json.loads(s, parse_float=Decimal)`

C. Cast afterwards

D. Not possible

3. What is JSON Lines good for?

A. Pretty printing

B. Streams and logs: one record per line, appendable and readable incrementally

C. Nested documents

D. Binary data

4. Which file mode does `tomllib.load` require?

A. `"r"`

B. `"rb"`

C. `"w"`

D. Any

5. What does `json.loads` guarantee about the parsed document?

A. That it has the shape your code expects

B. Only that the text was valid JSON — shape must be validated separately

C. That all numbers are ints

D. That keys are unique and sorted

ANSWER KEY · JSON IN DEPTH — CUSTOM ENCODERS, DECODERS, DATACLASSES AND CONFIG FILES

1. B Supply `default=` (or a `JSONEncoder` subclass) that converts unknown types.

2. B The hook receives the numeric text and builds the exact value; casting afterwards would already have lost precision.

3. B A single giant array cannot be processed until fully read; lines can.

4. B It reads bytes and decodes UTF-8 itself.

5. B Validate at the boundary with `match` or explicit checks.

LESSON 5 · 13 MIN

Bytes and binary data — struct, int.to_bytes, base64 and hashlib

Text is bytes plus an encoding; everything else a program reads from a socket, a device or a binary file is bytes plus a *layout*. Python's `bytes` and `bytearray` hold the raw data, `struct` packs and unpacks fixed layouts, `int.to_bytes` / `from_bytes` convert integers with an explicit byte order, `base64` turns bytes into text that survives a text channel, and `hashlib` produces the digests that identify and verify data. This lesson covers each, the byte-order question that decides whether a number reads as 1 or 16 777 216, and the binary-file idioms.

bytes and bytearray

```
b = b"hello"                # a bytes literal: ASCII only; other values as \x
b[0]                        # 104 — indexing gives an int
b[1:3]                      # b'el' — slicing gives bytes
bytes([104, 105])           # b'hi' — from ints 0-255
b.hex()                     # '68656c6c6f'
bytes.fromhex("68 69")      # b'hi'
b.decode("utf-8")           # 'hello' — to text
"héllo".encode("utf-8")     # b'h\xc3\xa9llo' — from text

ba = bytearray(b"abc")      # mutable
ba[0] = 65                  # bytearray(b'Abc')
ba.append(33); ba.extend(b"!!")
bytes(ba)                   # freeze
```

`bytes` supports most `str` methods on ASCII content (`split`, `strip`, `startswith`, `upper`, `find`) and `len` counts bytes. `bytearray` is the growable buffer for assembling messages; `memoryview(b)` gives a zero-copy window for slicing large buffers without duplicating them.

Byte order

The integer 1 stored in four bytes is `00 00 00 01` in **big-endian** (network order, most significant byte first) and `01 00 00 00` in **little-endian** (x86 and ARM). Reading with the

wrong order gives 16 777 216. Every format specifies its order; every conversion function takes it as an argument:

```
(1).to_bytes(4, "big")                # b'\x00\x00\x00\x01'
(1).to_bytes(4, "little")             # b'\x01\x00\x00\x00'
int.from_bytes(b"\x00\x00\x01\x00", "big")     # 256
(-1).to_bytes(2, "big", signed=True)  # b'\xff\xff'
int.from_bytes(b"\xff\xff", "big", signed=True) # -1
(255).bit_length()                    # 8 — how many bits a value needs
```

`to_bytes` raises `OverflowError` if the value does not fit; `signed=True` uses two's complement.

---

struct

`struct` packs several values into a fixed layout and back, with the order and sizes given by a format string:

```
import struct

record = struct.pack("<I4sH", 42, b"ab\x00\x00", 7)     # little-endian: uint
struct.unpack("<I4sH", record)                          # (42, b'ab\x00\x00',
struct.calcsize("<I4sH")                                # 10
struct.unpack_from(">H", buffer, offset=8)              # read at an offset wi
```

Prefix	Order	Code	Type	Size
`<`	little-endian, no padding	`b` / `B`	int8 / uint8	1
`>`	big-endian, no padding	`h` / `H`	int16 / uint16	2
`!`	network (big-endian)	`i` / `I`	int32 / uint32	4
`@`	native order and alignment (the default — avoid for files)	`q` / `Q`	int64 / uint64	8
		`f` / `d`	float32 / float64	4 / 8
		`Ns`	N bytes	N
		`?`	bool	1

Always give an explicit prefix: the native default inserts alignment padding that differs between machines. A record file is `pack` in a loop on write and `unpack` per `calcsize` chunk on read; `struct.Struct(fmt)` precompiles a format for repeated use and has `.size`.

Binary files

```
with open("data.bin", "wb") as f:
    for value in values:
        f.write(struct.pack("<i", value))

with open("data.bin", "rb") as f:
    while chunk := f.read(4):
        (value,) = struct.unpack("<i", chunk)
```

Binary mode reads and writes `bytes`, no encoding, no newline translation. `f.read(n)` returns fewer than `n` bytes only at the end; the walrus loop reads until empty. `f.seek(offset)` and `f.tell()` position the stream — random access into a record file is `seek(i * size)`.

base64

```
import base64
base64.b64encode(b"\x00\xff")           # b'AP8=' — bytes → ASCII bytes
base64.b64encode(b"\x00\xff").decode()  # 'AP8=' — as a str for JSON or a hea
base64.b64decode("AP8=")                # b'\x00\xff'
base64.urlsafe_b64encode(...)           # - and _ instead of + and /
```

Base64 is not encryption or compression; it is a way to put arbitrary bytes into a text-only channel (JSON, email, a URL) at a 33 % size cost. Binary data in a JSON document goes in as a base64 string.

hashlib

```
import hashlib
hashlib.sha256(b"hello").hexdigest()    # '2cf24dba5fb0a30e26e83b2ac5b9e29e1b
h = hashlib.sha256()
with open(path, "rb") as f:
    for chunk in iter(lambda: f.read(65536), b""):
        h.update(chunk)                 # hash a large file without loading i
h.hexdigest()
hashlib.md5(b"x").hexdigest()           # fast, non-cryptographic checksum us
```

A digest is a fixed-size fingerprint: the same input always gives the same digest, a different input almost certainly a different one, and the input cannot be recovered from it. Use SHA-256 for integrity checks and content addressing; MD5 and SHA-1 only for legacy checksums. Passwords are *not* hashed with `hashlib` directly — `hashlib.scrypt` / `pbkdf2_hmac` or a dedicated library with a salt and a work factor. `hmac.compare_digest` compares digests in constant time.

Pitfalls

- Indexing `bytes` and getting an int when a one-byte `bytes` was expected (`b[i:i+1]`).
- Forgetting the byte-order prefix in `struct` and getting platform padding.
- Reading a binary file in text mode (decoding errors, newline translation).
- Treating base64 as secrecy.

MD5 for anything security-related; a plain SHA-256 for passwords.

`str(b)` giving `"b'…'"` instead of decoding.

Key takeaways

`bytes` is immutable, `bytearray` mutable; indexing gives ints, slicing gives bytes; `encode` / `decode` cross to text.

`int.to_bytes(n, order, signed=)` / `from_bytes` convert integers; big-endian is network order, little-endian is the machine's.

`struct.pack` / `unpack` with an explicit `<` / `>` / `!` prefix lay out fixed records; `calcsize` sizes them.

Binary files: `"rb"` / `"wb"`, `read(n)` loops, `seek` for random access.

`base64` for bytes in text channels; `hashlib.sha256` for integrity, chunked `update` for large files.

EXERCISE 1

Packed records

Read records `name age score` and pack each with `struct` as `<8sHi` (an 8-byte name padded with NULs, a little-endian `uint16` age, an `int32` score) into `records_demo.bin`. Print the file size. Read the file back in fixed-size chunks with `struct.unpack`, printing each record with the NUL padding stripped from the name, then the SHA-256 hex digest of the file's bytes. Delete the file at the end.

Input: lines `name age score` (names of at most 8 ASCII characters). **Output:** `size <bytes>`, the records as `<name> <age> <score>`, then `sha256 <hex>`.

```
ada 36 -5
bob 20 100
```

prints

```
size 28
ada 36 -5
bob 20 100
sha256 be0a9402f32060565cc8920ff8e7a9f5685da720c79ffe7879e0646c9363736f
```

INPUT 1

```
ada 36 -5
bob 20 100
```

OUTPUT 1

```
size 28
ada 36 -5
bob 20 100
sha256
be0a9402f32060565cc8920ff8e7a9f5685da720c79ffe7879e0646c9363736f
```

REFERENCE SOLUTION

```python
import hashlib
import os
import struct
import sys

PATH = "records_demo.bin"
FMT = "<8sHi"
SIZE = struct.calcsize(FMT)
records = [line.split() for line in sys.stdin if line.strip()]
try:
    with open(PATH, "wb") as f:
        for name, age, score in records:
            f.write(struct.pack(FMT, name.encode("ascii"), int(age), int(score)))
    print(f"size {os.path.getsize(PATH)}")
    digest = hashlib.sha256()
    with open(PATH, "rb") as f:
        while chunk := f.read(SIZE):
            digest.update(chunk)
            raw_name, age, score = struct.unpack(FMT, chunk)
            print(raw_name.rstrip(b"\x00").decode("ascii"), age, score)
    print(f"sha256 {digest.hexdigest()}")
finally:
    if os.path.exists(PATH):
        os.remove(PATH)
```

EXERCISE 2

Byte orders and base64

For each integer on the input line print: its 4-byte big-endian hex, its 4-byte little-endian hex (both signed), the value read back from the big-endian bytes, the standard base64 of the big-endian bytes, and the SHA-256 hex digest of the integer's decimal text, truncated to 12 characters.

Input: one line of integers that fit in a signed 32-bit word. **Output:** `<n>: big=<hex> little=<hex> back=<n> b64=<text> sha=<12 hex>` per integer.

```
1 -1
```

prints

```
1: big=00000001 little=01000000 back=1 b64=AAAAAQ== sha=6b86b273ff34
-1: big=ffffffff little=ffffffff back=-1 b64=/////w== sha=1bad6b8cf971
```

INPUT 1

```
1 -1
```

OUTPUT 1

```
1: big=00000001 little=01000000 back=1
b64=AAAAAQ== sha=6b86b273ff34
-1: big=ffffffff little=ffffffff back=-1
b64=/////w== sha=1bad6b8cf971
```

REFERENCE SOLUTION

```python
import base64
import hashlib

for tok in input().split():
    n = int(tok)
    big = n.to_bytes(4, "big", signed=True)
    little = n.to_bytes(4, "little", signed=True)
    back = int.from_bytes(big, "big", signed=True)
    b64 = base64.b64encode(big).decode("ascii")
    sha = hashlib.sha256(tok.encode("ascii")).hexdigest()[:12]
    print(f"{n}: big={big.hex()} little={little.hex()} back={back} b64={b64} sha={sh
```

QUIZ

1. What is `b"abc"[0]`?

A. `b'a'`

B. `97`

C. `'a'`

D. `TypeError`

2. What does `(1).to_bytes(2, "big")` produce?

A. `b'\x01\x00'`

B. `b'\x00\x01'`

C. `b'1'`

D. `b'\x01'`

3. Why always give `struct` a byte-order prefix?

A. It is required syntax

B. The native default adds platform-dependent alignment padding and order

C. For speed

D. To allow floats

4. What is base64 for?

A. Encryption

B. Carrying arbitrary bytes through a text-only channel

C. Compression

D. Hashing

5. How is a large file hashed without loading it entirely?

A. `hashlib.sha256(f.read())`

B. Create the hash object and call `update` on successive chunks

C. It cannot be

D. `hashlib.file(path)`

ANSWER KEY · BYTES AND BINARY DATA — STRUCT, INT.TO_BYTES, BASE64 AND HASHLIB

1. B Indexing bytes gives an int; slicing `b[0:1]` gives `b'a'`.

2. B Big-endian puts the most significant byte first; little-endian would give `01 00`.

3. B `<`, `>` or `!` make the layout portable across machines.

4. B It is reversible and public; it costs about a third more space.

5. B The digest is the same as hashing the whole contents at once.

LESSON 6 · 14 MIN

sqlite3 — a SQL database in the standard library

Every Python installation ships a complete relational database: `sqlite3` wraps SQLite, a single-file (or in-memory) SQL engine used by browsers, phones and most desktop applications. It is the right tool when data has relationships, needs querying by several criteria, must survive a process, or is too big for memory — and it is the fastest way to learn SQL, because there is nothing to install. This lesson covers connecting, creating tables, inserting with parameters (never string formatting), querying with `fetchone` / `fetchall` /iteration, transactions and `with`, `row_factory` for dict-like rows, aggregation with `GROUP BY` and a join, and the mapping between Python and SQL types.

Connecting and creating

```
import sqlite3

con = sqlite3.connect(":memory:")          # or a file path: "app.db"
con.execute("""
    CREATE TABLE items (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL UNIQUE,
        price_cents INTEGER NOT NULL,
        category TEXT
    )
""")
```

`connect` opens (creating if absent) a database; `":memory:"` is a throwaway one for tests and judged programs. `execute` runs one statement; `executescript` runs several separated by semicolons (schema setup). `INTEGER PRIMARY KEY` auto-assigns ids. SQLite's column types are advisory (it stores what you give it), so keep money as integer cents and dates as ISO strings.

Inserting with parameters

```
con.execute("INSERT INTO items (name, price_cents, category) VALUES (?, ?, ?)"
con.executemany(
    "INSERT INTO items (name, price_cents, category) VALUES (?, ?, ?)",
    [("nut", 2, "hardware"), ("glue", 300, "supplies")],
)
con.commit()
```

The `?` placeholders are filled by the driver, which quotes and escapes correctly. **Never** build SQL with f-strings or `%` from values — `f"... WHERE name = '{name}'"` is the SQL-injection bug, and it also breaks on any name containing a quote. Named placeholders (`:name`, with a dict) are the alternative style. `executemany` is the fast way to insert many rows; `cursor.lastrowid` gives the id just assigned.

Querying

```
cur = con.execute("SELECT name, price_cents FROM items WHERE category = ? ORDE
cur.fetchone()                # ('bolt', 5) — a tuple, or None
cur.fetchall()                # the remaining rows as a list of tuples
for name, price in con.execute("SELECT name, price_cents FROM items ORDER BY p
    ...                       # iterate the cursor directly — the idiom
(count,) = con.execute("SELECT COUNT(*) FROM items").fetchone()
```

Rows are tuples in column order; iterate the cursor to stream, `fetchall` for a small result. Always add `ORDER BY` when the order matters — without it, SQL guarantees nothing, and a judged output must be deterministic.

Dict-like rows

```
con.row_factory = sqlite3.Row
row = con.execute("SELECT * FROM items WHERE name = ?", ("bolt",)).fetchone()
row["price_cents"], row[1], row.keys()          # by name, by index, the colu
dict(row)
```

`sqlite3.Row` gives name access with no per-row cost; set it on the connection once.

Transactions

```
with con:                                              # commits on success, rolls ba
    con.execute("UPDATE items SET price_cents = price_cents * 2 WHERE categor
    con.execute("DELETE FROM items WHERE price_cents > ?", (10_000,))
```

The connection as a context manager wraps a transaction: every statement in the block is committed together or not at all. One subtlety: `sqlite3` opens a transaction implicitly on the first modifying statement, so a rollback undoes *everything since the last commit* — including statements executed before the `with` block that were never committed. Commit the work you want to keep before starting a block that may abort. Without `commit()` (or the `with`), changes to a file database are lost when the connection closes. `con.close()` at the end — or `contextlib.closing(sqlite3.connect(...))`, since the connection's own `with` does not close it.

Aggregation and joins

```
con.execute("CREATE TABLE sales (item_id INTEGER REFERENCES items(id), qty INT
for category, total in con.execute("""
    SELECT i.category, SUM(s.qty * i.price_cents) AS revenue
    FROM sales AS s JOIN items AS i ON i.id = s.item_id
    GROUP BY i.category
    ORDER BY revenue DESC, i.category
"""):
    print(category, total)
```

`GROUP BY` with `SUM` / `COUNT` / `AVG` / `MIN` / `MAX`, `JOIN … ON`, `WHERE` before grouping and `HAVING` after, `ORDER BY` with tie-breakers, `LIMIT` — the core of SQL is small, and `sqlite3` is where to practise it. A query that would be three nested Python loops over lists is one statement, and the engine chooses the algorithm.

Types

Python	SQLite
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

Booleans go in as 0/1, dates as ISO strings (`date.isoformat()` in, `date.fromisoformat` out), `Decimal` as text or integer cents. The module's automatic date adapters are deprecated in 3.12; convert explicitly.

When sqlite3 is the answer

A script that accumulates results across runs; a dataset queried by several fields; a test that needs a real database; a desktop or mobile app's storage; a prototype before a server database. It is not for many concurrent writers or for a networked service — that is PostgreSQL or MySQL, reached through a driver with the same DB-API interface (`connect`, `execute`, `?` or `%s` placeholders), so what you learn here transfers.

Pitfalls

SQL built from values with f-strings.

A missing `ORDER BY` on output.

Forgetting `commit()` / `with con:` on a file database.

Floats for money.

`fetchall()` on a huge result when iterating would stream.

Assuming SQLite enforces column types (it does not, unless the table is declared `STRICT`).

Key takeaways

`sqlite3.connect(":memory:" | path)`; `execute` / `executemany` with `?` parameters — never string-formatted SQL.

Query with `execute(...)` and iterate, `fetchone`, `fetchall`; `ORDER BY` for determinism; `row_factory = sqlite3.Row` for name access.

`with con:` is a transaction: commit on success, rollback on error; `close()` separately.

`GROUP BY`, aggregates, `JOIN` and `HAVING` replace nested loops; the engine picks the plan.

Types are advisory: ints for money, ISO strings for dates, `None` for NULL.

EXERCISE 1

Inventory in SQLite

Keep an inventory in an in-memory `sqlite3` database with a table `items(name TEXT PRIMARY KEY, price_cents INTEGER, category TEXT)`. Commands: `add name price category` inserts or replaces (`INSERT OR REPLACE`, with `?` parameters); `report` prints `<category> <count> <total>` per category from one `GROUP BY` query ordered by category; `find name` prints the item's price and category or `missing`.

Input: commands. **Output:** the report rows and the finds.

```
add bolt 5 hardware
add nut 2 hardware
add glue 300 supplies
add bolt 6 hardware
report
find nut
find screw
```

prints

```
hardware 2 8
supplies 1 300
2 hardware
missing
```

INPUT 1

```
add bolt 5 hardware
add nut 2 hardware
add glue 300 supplies
add bolt 6 hardware
report
find nut
find screw
```

OUTPUT 1

```
hardware 2 8
supplies 1 300
2 hardware
missing
```

REFERENCE SOLUTION

```python
import sqlite3
import sys

con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE items (name TEXT PRIMARY KEY, price_cents INTEGER NOT NULL
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "add":
        con.execute("INSERT OR REPLACE INTO items (name, price_cents, category) VALU
    elif cmd == "report":
        for category, count, total in con.execute(
            "SELECT category, COUNT(*), SUM(price_cents) FROM items GROUP BY categor
        ):
            print(category, count, total)
    elif cmd == "find":
        row = con.execute("SELECT price_cents, category FROM items WHERE name = ?",
        print(f"{row[0]} {row[1]}" if row else "missing")
```

EXERCISE 2

Revenue by join

Two tables: `items(id INTEGER PRIMARY KEY, name TEXT, price_cents INTEGER)` and `sales(item_id INTEGER, qty INTEGER)`. Lines `item <name> <price>` insert items (ids assigned automatically); lines `sale <name> <qty>` insert a sale by looking up the item's id (`unknown item` if absent). At the end run one `JOIN` + `GROUP BY` query and print `<name> <units> <revenue>` per item that has sales, ordered by revenue descending then name, followed by `total <sum>`.

Input: lines. **Output:** the error lines as they occur, then the report.

```
item bolt 5
item nut 2
sale bolt 10
sale nut 100
sale screw 1
sale bolt 5
```

prints

```
unknown item
nut 100 200
bolt 15 75
total 275
```

INPUT 1

```
item bolt 5
item nut 2
sale bolt 10
sale nut 100
sale screw 1
sale bolt 5
```

OUTPUT 1

```
unknown item
nut 100 200
bolt 15 75
total 275
```

REFERENCE SOLUTION

```
import sqlite3
import sys

con = sqlite3.connect(":memory:")
con.executescript("""
    CREATE TABLE items (id INTEGER PRIMARY KEY, name TEXT UNIQUE, price_cents INTEGE
    CREATE TABLE sales (item_id INTEGER REFERENCES items(id), qty INTEGER);
""")
for line in sys.stdin:
    cmd, name, number = line.split()
    if cmd == "item":
        con.execute("INSERT INTO items (name, price_cents) VALUES (?, ?)", (name, ir
    elif cmd == "sale":
        row = con.execute("SELECT id FROM items WHERE name = ?", (name,)).fetchone()
        if row is None:
            print("unknown item")
        else:
            con.execute("INSERT INTO sales (item_id, qty) VALUES (?, ?)", (row[0], i
total = 0
for name, units, revenue in con.execute("""
    SELECT i.name, SUM(s.qty), SUM(s.qty * i.price_cents) AS revenue
    FROM sales AS s JOIN items AS i ON i.id = s.item_id
    GROUP BY i.id
    ORDER BY revenue DESC, i.name
"""):
    print(name, units, revenue)
    total += revenue
print(f"total {total}")
```

QUIZ

1. Why is `con.execute(f"SELECT * FROM t WHERE name = '{name}'")` wrong?

A. It is slow

B. It allows SQL injection and breaks on a quote in the name; use `?` parameters

C. SQLite does not support `WHERE`

D. f-strings cannot contain SQL

2. What does `with con:` do for a `sqlite3` connection?

A. Closes it

B. Wraps a transaction: commit on success, rollback on exception — without closing

C. Opens a cursor

D. Nothing

3. What happens to a file database's changes without `commit()`?

A. They are saved automatically

B. They are lost when the connection closes

C. They are saved on the next `execute`

D. SQLite raises

4. Why add `ORDER BY` to a query whose rows are printed?

A. For speed

B. SQL guarantees no row order without it

C. It is required by `sqlite3`

D. To enable `GROUP BY`

5. What does `con.row_factory = sqlite3.Row` enable?

A. Faster queries

B. Access to columns by name as well as by index

C. Automatic commits

D. Type conversion

ANSWER KEY · SQLITE3 — A SQL DATABASE IN THE STANDARD LIBRARY

1. B The driver quotes parameter values correctly; string formatting cannot.

2. B Close separately, or use `contextlib.closing`.

3. B `with con:` commits for you; explicit `commit()` otherwise.

4. B Unordered output can change between runs and versions.

5. B `row["name"]`, `row.keys()` and `dict(row)` become available.

CHECKPOINT · 25 MIN

Checkpoint — Files and data formats

This checkpoint covers the whole module: `open` with modes, encodings and `with`, line iteration and buffered writes, `pathlib` for building, querying and listing paths, CSV with `DictReader` / `DictWriter` and the quoting rules, JSON with custom encoders, decoders and dataclass round trips, bytes with `struct`, byte order, `base64` and `hashlib`, and `sqlite3` with parameters, transactions and aggregation.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why state `encoding="utf-8"` on every text `open`, and what does `"w"` do to an existing file?
- What is `Path("a") / "b.tar.gz"` `.suffix`, and how do you get both suffixes?
- Why does `csv` need `newline=""`, and what type is every field it reads?
- How do you serialise a `date` or a `set` with `json.dumps`?
- What does `struct.pack("<I", 1)` produce, and what would `">I"` give?
- Why must SQL never be built with an f-string, and what does `with con:` guarantee?
- Why add `ORDER BY` to a query whose output is printed?

The three programs are a CSV-to-JSON converter that writes a file and reads it back, a binary record store written with `struct` and verified with a SHA-256 digest, and an in-memory `sqlite3` report built from lines of input with a grouped, ordered query.

EXERCISE 1

CSV to JSON and back

Read a CSV document (`name,qty,price`) from standard input with `DictReader`, convert each row to a dict with `qty` as `int` and `price` as a string, and write the list as JSON (`indent=2`, `sort_keys=True`) to `items_demo.json` using `pathlib.Path.write_text`. Print the file's size. Read it back with `json.loads(path.read_text())`, print `<name> x<qty> = <qty * price>` per item with two decimals (use `Decimal` for the price), then `total <sum>`. Delete the file at the end.

Input: a CSV document. **Output:** `size <bytes>`, the item lines, `total <sum>`.

```
name,qty,price
bolt,3,0.50
nut,10,0.25
```

prints

```
size 128
bolt x3 = 1.50
nut x10 = 2.50
total 4.00
```

INPUT 1

```
name,qty,price
bolt,3,0.50
nut,10,0.25
```

OUTPUT 1

```
size 128
bolt x3 = 1.50
nut x10 = 2.50
total 4.00
```

REFERENCE SOLUTION

```python
import csv
import json
import sys
from decimal import Decimal
from pathlib import Path

path = Path("items_demo.json")
try:
    rows = [{"name": r["name"], "qty": int(r["qty"]), "price": r["price"]} for r in
    path.write_text(json.dumps(rows, indent=2, sort_keys=True), encoding="utf-8")
    print(f"size {path.stat().st_size}")
    total = Decimal("0")
    for item in json.loads(path.read_text(encoding="utf-8")):
        line_total = item["qty"] * Decimal(item["price"])
        total += line_total
        print(f"{item['name']} x{item['qty']} = {line_total:.2f}")
    print(f"total {total:.2f}")
finally:
    path.unlink(missing_ok=True)
```

EXERCISE 2

Binary record store

Write records `id score` packed as `>Ii` (big-endian `uint32` id, `int32` score) to `store_demo.bin`; print the file size and its SHA-256 hex digest. Then answer `get <index>` queries by seeking directly to `index * record_size` and unpacking one record (`missing` when the index is out of range). Delete the file at the end.

Input: `n`, then `n` record lines, then query lines. **Output:** `size <bytes>`, `sha256 <hex>`, then one line per query: `<id> <score>` or `missing`.

```
2
7 -3
9 42
get 1
get 5
```

prints

```
size 16
sha256 78f35ff4e3885e4f72f4593cda25699a3c3250086cfedc6ef28e4d03a05c8913
9 42
missing
```

INPUT 1

```
2
7 -3
9 42
get 1
get 5
```

OUTPUT 1

```
size 16
sha256
78f35ff4e3885e4f72f4593cda25699a3c3250086cfedc6ef28e4d03a05c8913
9 42
missing
```

REFERENCE SOLUTION

```
import hashlib
import os
import struct
import sys

PATH = "store_demo.bin"
FMT = ">Ii"
SIZE = struct.calcsize(FMT)
n = int(input())
records = [tuple(map(int, input().split())) for _ in range(n)]
try:
    with open(PATH, "wb") as f:
        for rid, score in records:
            f.write(struct.pack(FMT, rid, score))
    data = open(PATH, "rb").read()
    print(f"size {len(data)}")
    print(f"sha256 {hashlib.sha256(data).hexdigest()}")
    with open(PATH, "rb") as f:
        for line in sys.stdin:
            _, index = line.split()
            index = int(index)
            if not 0 <= index < n:
                print("missing")
                continue
            f.seek(index * SIZE)
            rid, score = struct.unpack(FMT, f.read(SIZE))
            print(rid, score)
finally:
    if os.path.exists(PATH):
        os.remove(PATH)
```

EXERCISE 3

Grouped report with a rollback

Build an in-memory `sqlite3` table `expenses(person TEXT, category TEXT, cents INTEGER)` from `add person category cents` lines, committing each one immediately with `con.commit()`. A line `batch-fail` starts a transaction with `with con:` in which the next lines up to `end` are inserted and then a `RuntimeError` is raised, so the whole batch is rolled back (print `rolled back`). Finally print `<person> <category> <total>` rows from a `GROUP BY person, category` query ordered by person then category, and `grand <total>`.

Input: lines. **Output:** `rolled back` lines as they occur, then the report.

```
add ada food 500
add ada travel 1200
batch-fail
add bob food 999
end
add bob food 300
```

prints

```
rolled back
ada food 500
ada travel 1200
bob food 300
grand 2000
```

INPUT 1

```
add ada food 500
add ada travel 1200
batch-fail
add bob food 999
end
add bob food 300
```

OUTPUT 1

```
rolled back
ada food 500
ada travel 1200
bob food 300
grand 2000
```

REFERENCE SOLUTION

```
import sqlite3
import sys

con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE expenses (person TEXT, category TEXT, cents INTEGER)")
INSERT = "INSERT INTO expenses VALUES (?, ?, ?)"
lines = [line.split() for line in sys.stdin if line.strip()]
i = 0
while i < len(lines):
    parts = lines[i]
    if parts[0] == "add":
        con.execute(INSERT, (parts[1], parts[2], int(parts[3])))
        con.commit()
        i += 1
    elif parts[0] == "batch-fail":
        j = i + 1
        try:
            with con:
                while lines[j][0] != "end":
                    _, person, category, cents = lines[j]
                    con.execute(INSERT, (person, category, int(cents)))
                    j += 1
                raise RuntimeError("batch failed")
        except RuntimeError:
            print("rolled back")
        i = j + 1
    else:
        i += 1
grand = 0
for person, category, total in con.execute(
    "SELECT person, category, SUM(cents) FROM expenses GROUP BY person, category ORD
):
    print(person, category, total)
    grand += total
print(f"grand {grand}")
```

CHECKPOINT QUESTIONS

1. What does `for line in f:` yield for a text file?

A. Lines without newlines

B. Lines with their trailing newline

C. Words

D. Bytes

2. Which `Path` method writes a string to a file in one call?

A. `Path.write()`

B. `Path.write_text(text, encoding=…)`

C. `Path.save()`

D. `Path.dump()`

3. What does `csv.DictReader` use as keys?

A. Column indexes

B. The first row's values (or `fieldnames=`)

C. Sorted column names

D. Nothing — it returns lists

4. What does `json.dumps(obj, default=f)` do with an unsupported object?

A. Skips it

B. Calls `f(obj)` and serialises the return value

C. Raises immediately

D. Uses `str(obj)`

5. Which is the correct dataclass-to-JSON round trip?

A. `json.dumps(obj)` / `json.loads`

B. `json.dumps(asdict(obj))` / `Cls(**json.loads(text))`

C. `pickle`

D. `str(obj)`

6. What does `struct.calcsize("<Ii")` return?

A. `4`

B. `8`

C. `2`

D. `16`

7. How do you read a binary file record by record?

A. `for line in f:`

B. `f.read(size)` in a loop until it returns empty bytes

C. `f.readlines()`

D. `json.load(f)`

8. What is `hashlib.sha256(b"").hexdigest()[:8]`?

A. `00000000`

B. `e3b0c442`

C. `ffffffff`

D. It raises

9. Which SQL placeholder style does `sqlite3` use?

A. `%s`

B. `?` (or named `:name`)

C. `{}`

D. `$1`

10. What does `INSERT OR REPLACE` do on a primary-key conflict?

A. Raises `IntegrityError`

B. Deletes the old row and inserts the new one

C. Ignores the new row

D. Merges the rows

11. Why keep money as integer cents in SQLite?

A. SQLite has no REAL type

B. Column types are advisory and floats carry rounding error; integers are exact

C. Integers are smaller

D. It is required by `GROUP BY`

12. What happens to a `"w"`-mode file if the program crashes halfway through writing?

A. The old contents are restored

B. The file holds partial data — write a temporary file and `os.replace` it to be safe

C. Nothing is written

D. The file is locked

ANSWER KEY · CHECKPOINT — FILES AND DATA FORMATS

1. B Strip with `rstrip("\n")` when the newline is unwanted.

2. B It creates or truncates the file; `read_text` is the inverse.

3. B Each subsequent row becomes a dict keyed by the header.

4. B `f` must return something JSON can encode, or raise `TypeError`.

5. B Dataclasses are not JSON-serialisable directly; `asdict` produces the plain dict.

6. B A `uint32` plus an `int32`, no padding with an explicit byte-order prefix.

7. B Binary files have no lines; fixed-size reads (or `seek` for random access) are the idiom.

8. B The empty input has a well-known digest beginning `e3b0c442…`; any input has a fixed-size digest.

9. B Pass the values as a tuple (or a dict for named placeholders).

10. B `INSERT OR IGNORE` would keep the old row; a plain `INSERT` raises.

11. B The same reasoning as in Python: exact arithmetic for money.

12. B Truncate-then-write loses the original on failure; rename-over is atomic on most systems.

CodeKairo · Python study plan · Module 14: Files and data formats. Solve the exercises online to track your progress and earn XP.